MASTER IN COMPUTER SCIENCE

Big Data for Automatic Relation Extraction in Natural Language Processing

Using Word Embedding and Word2vec

Master Thesis

Jérémy Serre

Supervisors

Dr. Prof. Philippe Cudré-Mauroux
Alisa Smirnova

eXascale Infolab
Université de Fribourg

November 2017

UNIVERSITÄT BERN

UNIVERSITÉ DE NEUCHÂTEL

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Abstract

Extracting relations from an unlabelled raw corpus is a complex task. Moreover, since the emergence of big data and its growing importance, the scientific world has to adapt and create tools to process larger and larger volume of data.

This master thesis work aims to improve the relation extraction task using the latest tools and technologies in the field of natural language processing and big data (like Word2vec and Spark). At this end, we will first define a framework that starts with a plain text and produces word embedding using the Word2Vec models. Word embedding project words from the corpus into a multi-vector space and thus, allow to perform vector operations in order to extract semantic relations. The objective of this project is to be able to rely on a relation between two words in order to extract new relations of the same type (e.g.: «Paris - France» in input, to extract new city - country relations like «Bern - Switzerland»). Moreover, we propose new methods for selecting the best relation pairs to use as input in order to improve the precision of the results and decrease the execution time. Furthermore, we propose a method using knowledge bases, to automatically evaluate the extracted relation pairs.

Acknowledgments

I would like to take this opportunity to thank all the people who have contributed in to this thesis, particularly Dr. Prof. Philippe Cudré-Mauroux and the University of Fribourg with eXascale Infolab which initiated the project.

I would also like to thank Alisa Smirnova who supervised and mentored my work, for her attention, and for all the time she granted me, her advice was really useful and appreciated.

Then, I would like to thank all the members of eXascale Infolab for their patience and the time they spent to help me in many ways.

Contents

List of Figures

List of Tables

Abbreviations

YARN Yet Another Resource Negotiator

HDFS Hadoop Distributed File System

KB Knowledge Base

nDCG normalized Discounted Cumulative Gain

SPARQL SPARQL Protocol and RDF Query Language

UML Unified Modeling Language

JVM Java Virtual Machine

RDD Resilient Distributed Dataset

NLP Natural language processing Introduction

Introduction

This project aims to create a framework that clearly explains each stage from the raw corpus extraction and processing to the extraction of relation and their evaluation.
Extracting relation from an unlabelled raw corpus is a complex task involving Natural Language Processing. We propose to use the word embeddings technology to solve this task, more specifically by using Word2Vec models. Word2Vec is a word embedding technic that is very efficient to find semantic/syntactic relations between words coming from a raw text corpus. In effect, transforming words into vectors allows to perform mathematical operations that can be very interesting in order to extract relations between words.

Most of the existing algorithms that use Word2vec models are only able to run on a single machine. However, machine learning models necessitates a lot of data and may take days to train. Thank to the growing of the big data, a lot of new tools have been created to help process these large volume of data. The latest technologies in the Big Data field like Spark allow to quickly and efficiently process large volume of data using distributed systems. We implemented all our solutions using a technology that allow to distribute the data and the calculations.

We also propose a method to automatically evaluate the extracted relations using a Free Knowledge Database (KB). Moreover, this thesis is also focused on maximising the number of « good » relations output, with as few pairs as possible in input. To this end, we propose four different methods for the input pairs selection.

The purpose of this master work is to solve all these problems, explain each step of the proposed solutions in the simplest manner, and implement each solution.

Objectives

This thesis aims to complete various objectives in the extraction relation field. The relation extraction part of this project started by analyzing an already existing program created by Matúš Pikuliak. This program uses the Gensim framework and was designed to run on a single machine (local mode). Therefore the first objective was to deploy this program on Spark in order to make it operable in a distributed environment and by replacing Gensim with Spark and its MLLIB library we could compare the results from the two methods. We also wanted to create methods to obtain better results with fewer pairs in input. We expect that using fewer pairs as input will improve the computation time and the results. Finally, we want to measure the precision of the relations extracted from the corpus, with the help of a Knowledge Base.

This master thesis will therefore detail the various useful steps to clean a raw text, train and fit a Word2Vec model with Spark. Finally, we present methods that help select the best input pairs, extract similar relations and evaluate them.

1. Prerequisite

This chapter is briefly presenting the main technologies used for this project.

1.1. Apache Spark

Working with big data sets requires a large amount of computational resources and execution time. Hence the advantage of using distributed storage systems like HDFS is to be able to save big data sets and reuse them for different calculations.

Apache Spark is an open source cluster computing framework [1]. It was implemented in Scala but it runs with JVM. Moreover Spark has an application programming interface for different languages Java, Python, Scala and R. It can work in standalone applications on cluster mode coupled with YARN if the data are stored in Hadoop 2.0. However, we can also run it in a local mode.

Spark provides a new data structure called the resilient distributed dataset (RDD), which is the basic abstraction. It represents an immutable, portioned collection of elements that can be operated on in parallel [2][3] with a fault tolerance. The idea behind this is to extend the limitation of the MapReduce (e.g. Hadoop), which works in steps instead of Spark which can work on all data at once. But Spark does not have a File System instead of Hadoop, but it can run on Hadoop Distributed File System.

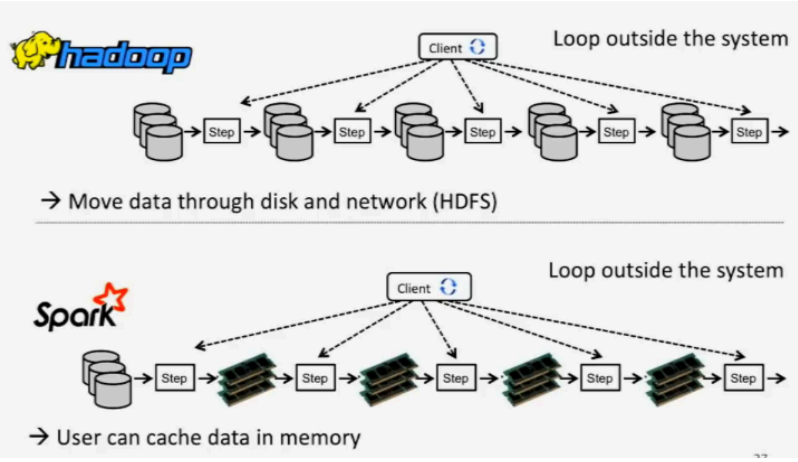

Figure 1: Hadoop and Spark system [4]

Spark Core provides distributed task dispatching, scheduling, and basic I/O functionalities using interface centered on the RDD abstraction. This interface is a functional model programming. There is a driver program invokes parallel operations such as map, filter or reduce on an RDD by passing a function to Spark, which then schedules the

function's execution in parallel on the cluster. All operations like joins, use RDDs in input and use it to produce a new one. Moreover resilient distributed datasets operations are lazy.

1.2. Word2vec

Word2Vec was created by a team of researchers led by Tomas Mikolov at Google. The algorithm was implemented by many other programmers in many languages. This algorithm produce word embedding [5][6] which is intended to represent each word as vector from data sets. Moreover Word2Vec provides two different structure (see in 1.2.2) to create models. Each of theses models have two-layer neural networks that are trained to reconstruct linguistic contexts of words.

1.2.1. Word Embedding

Word Embedding is Machine Learning feature from deep learning for Natural Language processing, where words from corpus are mapped to vectors of real numbers. So the idea is to project words to multi-dimensional space (vector space) in order to create mathematical operations. So this feature learning technique uses two-layer neural network and each of this neuron corresponds to one coordinate in the multidimensional space. Moreover we can obtain some connection without providing any information about its semantic. The most famous example of this vector word computation is « King - Man + Woman = ? » we obtain Queen.

1.2.2. Skip Gram and CBOW

There are two existing algorithms to train Word2Vec models. As we can see in the figure 2, we have three parts in each model training architecture. W corresponds to the selected Word and W(t-x) corresponds to a word from the context. If we define a window size of 5, the algorithm will, for each word, select the 2 preceding and following words in the sentence.

The main differences between these two algorithms are:

- Continuous bag-of-words (CBOW) model learns each word using its context. It can be summarized as follows, given a context, what is the probability of finding a word. As we can see in the Figure2, the context is used as input. If we have a sentence like « Word Embedding is Machine Learning feature from deep learning » and we want to predict the word, « Machine ». CBow will use the window size to define the words context. Example the context of « Machine » is « Embedding (t-2) is (t-1), learning(t+1), feature(t+2) », if the

window size is 5. So during the training part if these 4 words appear together as input the CBOW model has to predict, « Machine ».

- Skip Gram does the opposite, for each word the algorithm has to guess the context [7]. A word is given as input and we obtain the words that are likely to appear in the word context with respect to the window size. If we use the same example as described in the CBOW presentation, given the word « machine » we end with the 4 words surrounding it.

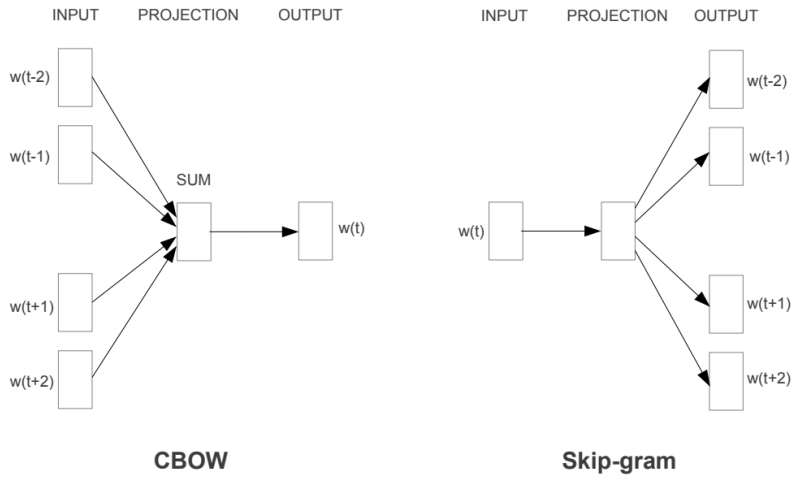

Figure 2: Schema of the two Word2Vec algorithms

As we know, each of these models has two neural layers. So in figure 3, we can see these two layers for the skip gram model. In this example we have a vocabulary of 10 000 words and 300 neurons. Each word is represented by 0 or 1, 1 is the hot vector (vector used). In this case the word is « ants », so it represents by 1 and all the other vectors by 0. The hidden layer is used like a weight matrix. The output layer use softmax [8] classifier because this function is useful to handle multiple classes and obtain a binary result. So, the output layer use an « activation function ».

Furthermore, during the training part, we have to adjust all of the neuron weights (hidden layers) slightly, so that it predicts that training sample more accurately(output layer).

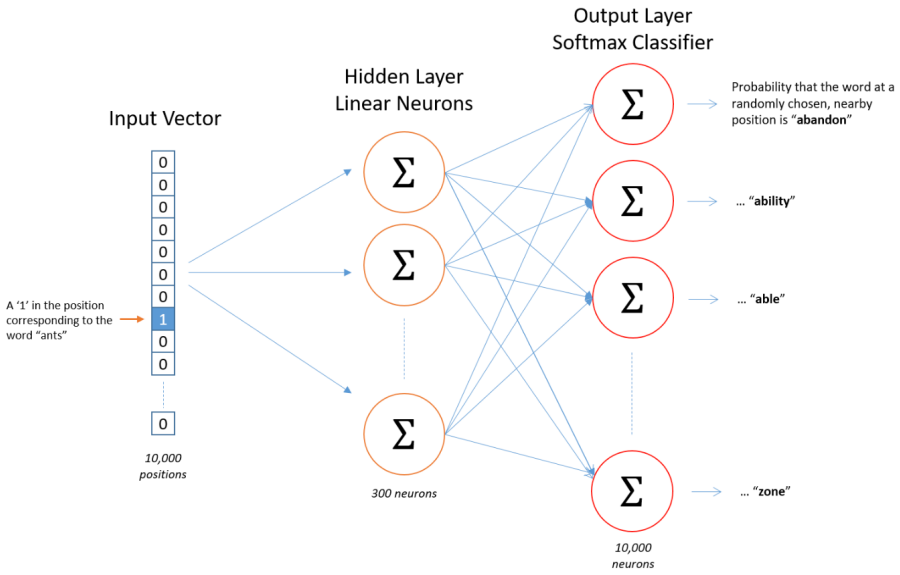

Figure 3: Skip Gram Neural Network Architecture

1.2.3. Hierarchical Softmax & Negative sampling

These algorithms will both be used in order to optimise the computation time during the gradient descent phase, which consists to adjust weights of a neural network.

The hierarchical softmax uses a binary tree representation of the output layer with the W words as its leaves and, for each node, explicitly represents the relative probabilities of its child nodes. This defines a random path that assigns probabilities to words. The idea behind is to limit the amount of output vectors that must be updated per training instance. In the Figure 4, we can see an example of a path from the root to **w2** which is highlighted. More precisely, each word **w** can be reached by an appropriate path from the root of the tree. This structure has a direct effect on the performance. In this case we use the binary Huffman tree, as it assigns short codes to the frequent words, which results in fast training. [9]

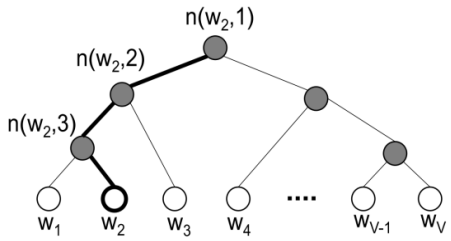

Figure 4: An example of a binary tree for the hierarchical softmax. The white units are words in the vocabulary, and the dark units are inner units.

Negative Sampling is simply the idea that we only update a sample of output words per iteration. The target output word should be kept in the sample and gets updated. Moreover we have to add to the sample some selected words as negative samples using an unigram distribution (words occurrence). These negative output words need to be trained to output 0. Indeed, if the weight of a word is reduced or increased and the output doesn't change, it can still influence the output of many other words. So the final idea is for each word in input, to just modify a subset of words instead of all the words. Because again one modification of one word weight can influence the whole network. Notice, the probability for selecting a word as a negative sample is related to its frequency, with more frequent words being more likely to be selected as negative samples.

1.2.4. Gensim and MLLIB

Gensim [10] is a python framework used for many tasks in NLP and it also including an implementation of Word2Vec. However, it is limited to run on a single machine whereas MLLIB can run on distributed data-parallel system. MLLIB comes from the framework Spark [11] which contains many machine-learning algorithms, including Word2Vec. Notice that these two frameworks use subsampling for frequent words to decrease the amount of training examples.

In this paper we will use MLLIB to create Word2Vec model. But MLLIB can only use the skip gram algorithm, and the Hierarchical Softmax instead of the Negative sampling algorithm [12], which is used to reduce the computation time for updating the weights.

1.3. WikiData

Wikidata is a Free Knowledge Database, more precisely a document-oriented database for Semantic Web. It is a collection of articles made up of data and key-value pairs, especially links to other articles, thus forming a semantically structured set of graphs. The ambition is to form a unique knowledge graph [13].

In this project, we use Wikidata to extract relations with SPARQL and send queries to the database through Wikidata query [14]. The goal is to get a JSON file and to compare it to our results in the evaluation process. In the figure 5, we can see an example of a query issued to obtain the Capital-Country relation. The **"ID"** contains the link of the subject, **"sub"** corresponds to the subject (which is the country), and **"obj"** is the object (in this case, the capital) as we can see in the figure 6

```
SELECT ?id ?sub ?obj WHERE {
    ?id wdt:P31 wd:Q6256.
    ?id wdt:P36 ?capital.
    ?id rdfs:label ?sub filter (lang(?sub) = "en")
    ?capital rdfs:label ?obj filter (lang(?obj) = "en")
}
ORDER BY ?sub
```

Figure 5: Query in SPARQL

Items are only identified by a **"Q"** followed by a number, such as country (**Q6256**). Properties in Wikidata have a **"P"** followed by a number, such as with capital (**P36**). Both can return values. **wdt** is used for truthy assertions about the data, links entity to value directly (predicates). And **wd** is used to return Wikibase entity.


```
[{"id":"http://www.wikidata.org/entity/Q889","sub":"Afghanistan","obj":"Kabul"},
{"id":"http://www.wikidata.org/entity/Q222","sub":"Albania","obj":"Tirana"},
{"id":"http://www.wikidata.org/entity/Q262","sub":"Algeria","obj":"Algiers"},
{"id":"http://www.wikidata.org/entity/Q228","sub":"Andorra","obj":"Andorra la Vella"},
```


Figure 6: JSON file extract from Wikidata with our structure

2. Approach

In this part we explain the main steps of the project and the code [15]. The major improvements brought by this Master's thesis are:

- Adapting an existing project to make it work using an MLLIB model on Spark.
- Replacing specific parts of the code with RDD operations, especially for the pairs generation part.
- Simplifying the program.
- Understanding and explaining how the existing code works.
- Operating in a distributed environment with Spark and HDFS.
- Creating a script that converts a Spark model to a Gensim one. This conversion aims to solve the compatibility problem and be able to compare the models.

- Creating methods for selecting the best pairs to use in input, in order to obtain better results with fewer input pairs.
- Creating a script to automatically evaluate the extracted relations, with a knowledge base.

2.1. Pre-processing

The first step is the selection of the text corpus that will be used to create the Word2vec model. The text inside the corpus will influence the model, if the text contains only financial articles, the model will not predict connections external to this field. In this thesis, we have selected a corpus that is meant to be general, i.e « enwiki dump » from Wikimedia dump [16]. Training a model with texts from various fields produces a versatile model.

The idea is to extract this corpus as a xml file and a convenient way to do that is to use the corpora.wikicorpus function from the Gensim Framework. Each article in the wiki corpus is formatted in a xml format. There is a need to parse these xml structures to only process and extract the individual words. The upper case words are replaced by a lower-case version in order to avoid word duplicates with different formats. A similar transformation is also performed on the words with accent(s), similarly to the upper case problem, we have decided to replace all the accent letters by the same letter without an accent. Furthermore, there are a lot of non-ASCII characters and the script should remove all of them as well. There are several ways to clean a text corpus and it depends on the final goal we want to achieve. However, for a higher precision of the model, all unnecessary words should be removed (see 2.1.2) or replaced as well.

The script 1_process_wiki.py was created for this purpose. To launch this script, one must call the script like this: « python3 1_process_wiki.py enwiki.xxx.xml.bz2 wiki.en.text ». The first argument is the wikimedia dump file name (ex: enwiki.xxx.xml.bz2) and the last argument is the name to give to the cleaned corpus file.

2.1.1. N-Gram

N-Gram is the name given to a word sequence that is composed of 1 or more words. Simply speaking it is used to describe if a word is composed of several parts or not. For example, the words « new » and « york » are uni-gram because they are composed of one word. But each of them can be used in different contexts, indeed when they are associated we obtain a new word. In our corpus we handle the bi-gram (composed of two words), tri-gram and quadri-gram. All the N-Gram are transformed into a single word, to do this we add one or more underscore(s) between the words, for example « new york times » is converted to « new_york_times ». The following formula was introduced in the paper [12] and it allows to compute a score used for validating or not a bigram. A high score means that the occurrence of the two words together is high compared to their occurrence alone. Therefore a high score increase the probability to validate a bi-gram.

$$score(w_i, w_j) = \frac{(count(w_i, w_j) - minCount) \times N}{count(w_i) \times count(w_j)}$$

We have to define a threshold value and when a bi-gram score is higher than the threshold, it is validated. Moreover, we have to set a « min count » which refers to the number of occurrences of these two, three or four words together. As we can see in the formula, **wi** and **wj** refer to the first and second words and **N** is the total vocabulary size in the paper. Although this formula only works for two inputs, it is possible to provide an n-gram for the arguments **wi** and/or **wj**.

2.1.2. Stopword Lists

A stop word list is a list containing the most common words of a language. These words are often associated with many other words and so are not useful in the project context. For example, the word « The » is in the stop word list.

The advantages of removing the words from a stopword list are:

- Reducing the size of the corpus.
- Avoiding a problem for the n-gram, because there is a high probability of having a lot of n-gram composed with these words.
- In Word2Vec we have to define a window size (described in 1.2.2) which refers to the context of an embedding word. So, if we remove these un-useful words, we only obtain a context composed of useful words and therefore a better representation in the vector space.

2.2. Fitting the model

This step is about fitting and training the models, in this case we want to fit Word2Vec models with Mllib in Spark. This is an important step because each modification has an impact on the final results. Moreover we have to define the number of artificial neurons also named "vector size", which represents the number of coordinates of the words in the vector space. Increasing the number of vectors does not necessarily result in a better model. Another important thing to set is the window size, which is usually 5 or 10 and the learning rate which is defined between 0.025 or 0.05. The script to use is 2_Mllib_Save_Load_Spark.py.

2.2.1. Parameters

To begin, we have to select the parameters to train the Word2Vec model. This is an important step to adjust the model as desired.

- Vector Size: By default it is set to 100, but most of the time it is between 100 and 300. This number represents the number of neurons in the neural network, so it impacts the computation time. Moreover, increasing this number does not necessarily increase the accuracy. A good way to test it is to start with 100, then 150, etc. and evaluate model results. If the results are almost similar, it is better to keep the lowest value because it is the fastest one. In this project we chose to set a vector size of 100.
- Learning Rate (default: 0.025): The learning rate is how quickly a network abandons old beliefs for new ones. The idea is to find a learning rate that is low enough so that the network converges to something useful, but high enough so that you don't have to spend lots of time to training it. Most of the time with words embedding it is set between 0.025 and 0.05. In our case we chose 0.05 because we obtained better result. Unfortunately, as it is often the case there is no global optimal value. It is depending on the context in which the model will be used, we have to make several tests with the value that improve the model results. It is also depending on the corpus, the reprocessing phase, etc.
- Number of Partitions (default: 1): This parameter is useful to considerably reduce the computation time to create a model, using a cluster. We chose to set this value to 4 because we obtained good results without affecting the accuracy of the model. This is often a trade-off between time computation and accuracy. It can be noted that a larger and more varied corpus allows us to set a higher partition number than with a small corpus, that will be much more impacted by partitioning.
- Number of Iterations (default: 1): First of all, this number has to be smaller than the number of Partition. Moreover increasing the number of iterations usually improve the quality of the word representations. In this case, the number of Partitions is low, so we don't change the default value. Furthermore, increasing this value affects the computation time.
- Min Count (default: 5): This parameter is useful to remove the « hapax legomena » which are the words that only occur once and the infrequent words. So, if it is set to 5, all the words that appear less than 5 times will be removed. Most of the time it is set between 5 and 10. Increasing this value reduces the computation time, because we have fewer words to handle. As we know, if we follow the Zipf's law and we create a Rank-Frequency plot, we can see that the number of words that occurs fewer than 10 times is higher than words with higher occurrence. We don't modify this value.

- WindowSize (default: 5): This parameter is the distance between the predicted and current word in a sentence. This value is usually set between 5 and 10. It is useful for the context of the predicted word, it's that why we don't have to choose a too high or low a value in order to obtain a good accuracy. Moreover, a higher value increases the computation time. In our case we set it to 10.

2.2.2. Training and Fitting

It is a complex task to choose the optimal parameters, a good solution is to create a lot of models each time modifying a single parameter. With this solution, we see the impact of a single parameter on the model, and adapt it in consequence. Moreover, more accuracy means more iterations, more iterations means more partitions and finally more partitions means less accuracy. It is why there is no single perfect solution. In this project, our objective is to create a Word2Vec model in a quick way with a good accuracy and this setting correspond to our needing. Moreover with spark, the model created use skip grams structure with hierarchical softmax to update the weights.

2.2.3. Model

We start by creating a Word2Vec instance which contains all the parameters needed. This instance is then used to fit the model. The fitted model has to be tested and a good way to do that is to try to extract different relations and see if the results are more or less what we expect.

```
############################
# Create the spark context #
############################

sc = SparkContext (appName="Build_MLLIB_MODEL_App")

####################
# Create the model #
####################

inp = sc.textFile(copusPath).map(lambda row: row.split(« »))

topNum = 20
seed = 42
itera = 1
alpha = 0.05
minCount = 5
window = 10
numPart = 4

# SET The word2vec model

word2vec = Word2Vec().setVectorSize(k).setSeed(seed).setNumIterations(itera).setLearningRate(alpha)
                          .setNumPartitions (numPart).setMinCount(minCount).setWindowSize(window)

model = word2vec.fit(inp)
```

Figure 7: Script to train and fit Word2Vec model with settings[15].

In the figure 7, we can see how to set all the parameters, the variable **word2vec** is an instance of word2vec and the value returned after ***word2vec.fit(inp)*** is the fitted model. Notice that ***inp*** contains all the corpus split by space.

2.3. Relation Extraction

The relation extraction is the most important part of our thesis. The objective is to extract new pairs with a similar semantic relation. The idea is to give in input, pairs with a similar relation like "A is part of B", or "A is the capital of B", etc.; and use these pairs to extract new pairs with the same relation. The quality of the results is depending on different factors such as the model, the semantic relation between the words, the original corpus. Some relations are easier to extract and produce better links. To see more explanation about how it works, please refer to the Chapter 3. The script for the relation extraction is run.py.

2.4. Evaluation

The last part is the evaluation of our results. We first have to query **wikidata** in order to obtain a JSON file of the evaluated relation. When we have this file, we can verify which relations are true or false and add annotations to text files. The idea is to validate the relations, for example using 0 or 1. However, in complex connections like masculine to feminine words, the validation may not be simply binary and the annotation should be adjusted by hand. The aim is to evaluate the extracted relations like in an information retrieval system. At this end we use the **normalized discounted cumulative gain(nDCG)**. Moreover, it is very interesting to use the nDCG score, because the extracted relations are ranked from most likely to least likely to be a good candidate. Indeed, the nDCG score takes into account if it is a good candidate and its rank in the output list. So, if all the good candidates are in the top of the list, we obtain a high nDCG score even if there are a lot of bad relations after them. Before computing the nDCG score, we have to compute to other formula: the **Discounted Cumulative Gain (DCG)** and the **ideal Discounted Cumulative Gain (iDCG)**.

$$\mathrm{DCG_p} = \sum_{i=1}^{p} \frac{rel_i}{\log_2(i+1)}$$

The DCG and the iDCG formulas are almost similar except for the rank order, in effect the iDCG formula sorts in descending order. **p** is the number of relations extracted and **rel** corresponds to the score of the relation **i**.

$$\mathrm{nDCG_p} = \frac{DCG_p}{IDCG_p}$$

To summarize, we used various methods to evaluate the model. The nDCG formula was used to measure the ranking quality, we also count the number of good relations and compute the precision. We can select the number of relations to be output: if 10 is selected, the precision of the semantic relations will be high, whereas if 100 is selected, the precision will be lower. That is why we use two methods, one that take into account the rank of the good candidates and another that is not considering the rank but only the proportion of good candidates. A script was made for the evaluation part with an implementation of the nDCG score in 4_evaluate.py.

2.5. Additional features

One of the additional features is to solve the problem of compatibility between a model created by Mllib and a model created by Gensim. So the script « Optional_Millib_Model_To_Gensim_Model.py » was created to transform a model from Mllib to Gensim and keeping the same structure. The idea behind this transformation process is to test if there is a difference between the original program developed to work locally and the second in Spark, with a same model. In Mllib the model was saved in « snappy parquet » and we have to translate it to. txt or. bin with the correct structure.

Another additional feature is a script « visualization.py », that was created to obtain a 2D visualization of our models [17]. This feature is interesting to analyze our model after the training part by seeing if similar elements are close in the vector space.

3. Input Pair Selection

3.1. Introduction

In this part, we present 4 methods for selecting the pairs to use as input. Our algorithm is, given pairs of the same relation type in input outputting new pairs of the same relation type. One of the objectives is to obtain a high precision with fewer input pairs as possible. We later compared the 4 methods with different relation types, using a list of 25 input pairs and also by selecting a subset of 5, 10, 15, 20 elements from this list. The figure 8 is a schema of the Pair structure. The pair structure is composed of 3 embedding objects. Each embedding object is composed of one word and its vector representation in multi-vectorial space.

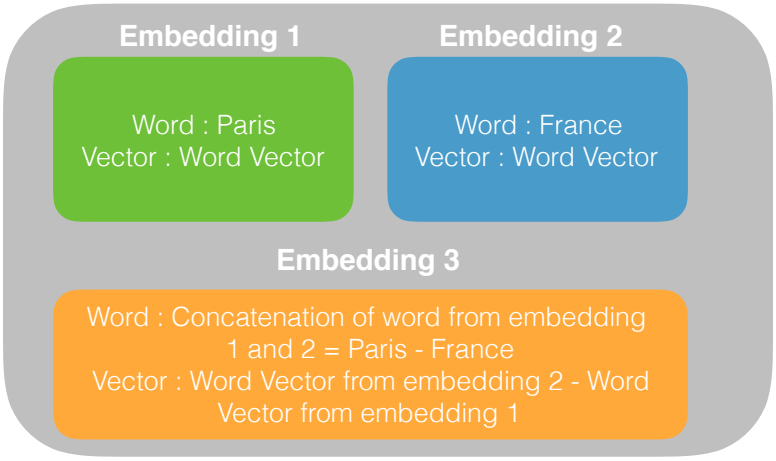

Figure 8: Schema of Pair structure

The input pair selection part is very important for the relation extraction. The idea of tuning the input pairs selection came after having analyzed how the relation extraction algorithm works (see 4.4). During the generation of new pairs, the relation extraction algorithm uses a Word2Vec model function to retrieve a list of the most similar words to a given word. For example: given the word "Paris", the Word2vec model function will return a list that can be composed of other capitals (like western European capitals, Madrid, London, etc.). In a similar way, for an input composed of pairs all representing the same relation as « Capital-Country » (Example: Paris - France", "Madrid - Spain", "London - England » for western countries) the model will also output new pairs of the same relation type. Since the word vectors from the pairs of the same relation type are close, we explored the possibility to select one pair that is best representing the entire pairs list. This concept is used only for Pair Cosine & Euclidean selection plus K-Means Selection.

The figure 9 explains the main steps of the 4 methods explained before. As we can see at the begin the user give in input a set of pairs, the global idea (except for word count method) is to cluster (or group) the pairs with something in common together. The final idea is to obtain groups which contains pair, and we have to rank these pairs inside the group in order to sort from the best to the least good representative of the group. The last part is the selection of the pair in each group, to more details see figure 11.

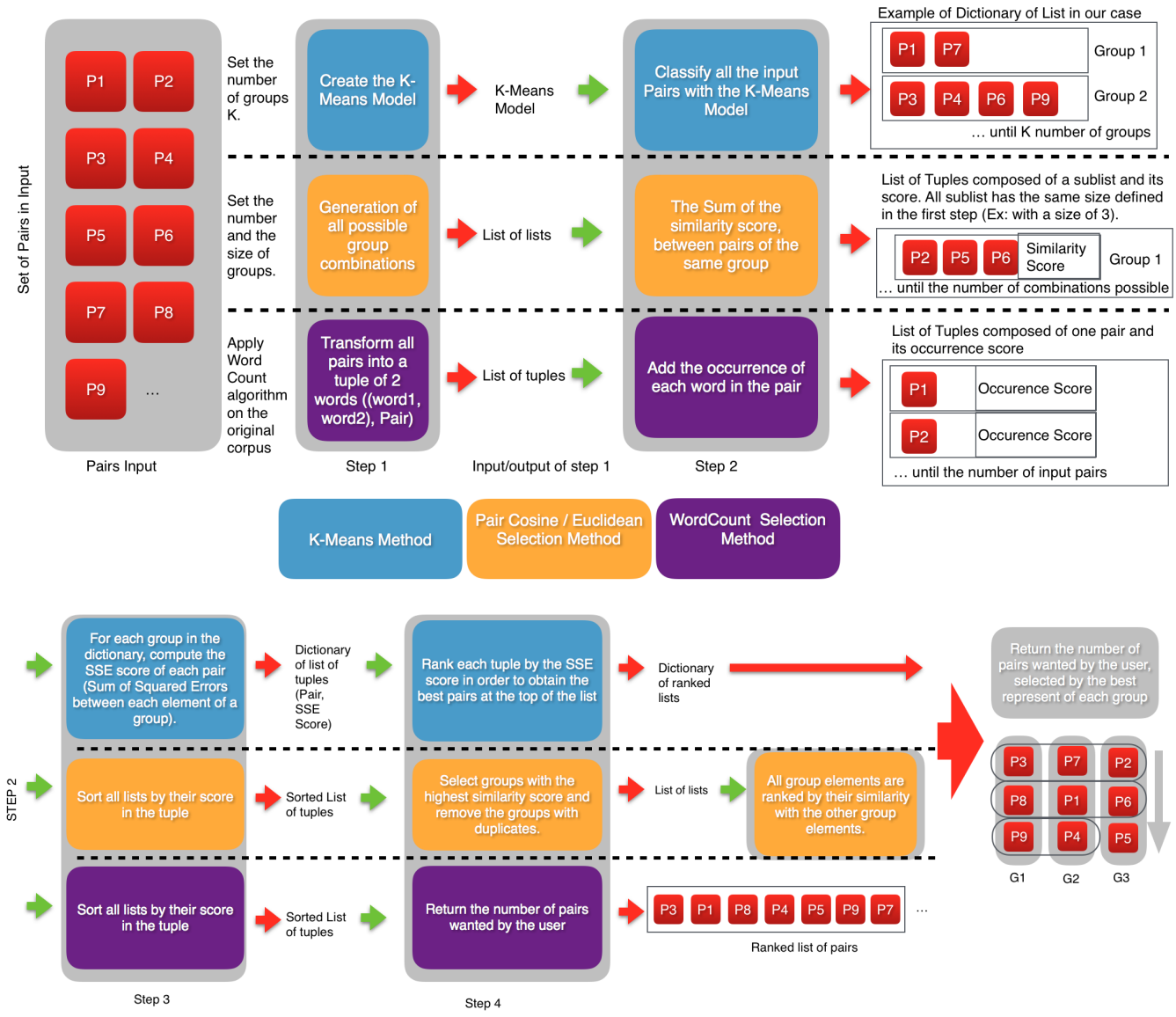

Figure 9: Input Pair Selection schema for the four methods divide in steps. The first line with blue box is for K-Means method, in the second line in orange both the method Pair Cosine and Euclidean Selection and the last line in purple Word Count method. The red square represent a pair of this schema.

3.2. WordCount

WordCount is a simple method, that only count the occurrence of each word in the pairs, and sum them to obtain the occurrence score of a pair. The same operation is done for 25 elements and they are at the end, ranked by their scores, in descending order. The pairs are selected amongst the top ones. This method is only used as a basis to later compared with more complex selection methods.

3.3. Pair Cosine Selection & Euclidean Selection

The idea of the "Input Pairs Selection" is to group the most similar elements together, and select the groups with the elements closest to each other. In addition, one representative is selected for each group. This representative is the element that share the most "commonalities" with the other elements of its group. This grouping of similar elements is a kind of clustering method. The similarity function used is the only difference between the Pair Cosine Selection method and the Euclidean Selection method. Indeed, the first method uses the Cosine similarity and the second method uses the Euclidean similarity.

Here are the details of the pairs selection methods:

- First, we have to set the number of groups we want. Then, all the possible combinations are generated in a « brute force » fashion. Inside our program all these operations are performed using the Spark RDD structure. For example, if we provide 25 input pairs we can produce 5 groups of 5 elements each.
- The presented formula is calculating, « how many groups are generated ». **K** is the number of elements by combinations and **N** is the number of different values. Continuing with our example, **K** = 5 because we want to obtain groups of 5 elements and **N** is equal to 25. The result is 53130 unique groups of 5 pairs. All the combinations do not contain duplicates (ex: [A, B, A] is not valid), and the order is not taken into account ([A, B, C] = [A, C, B] only one of these will be retained).

$$C = \frac{N!}{K!(N-K)!}$$

- The idea is to compare in each group, each element to the other elements within the group, with either Cosine Similarity or Euclidean Similarity (Euclidean distance).
- The next step is to average all these scores to get the similarity score of each group.
- Then, we choose the groups which contain the most similar elements, and which have a better score. Once a group is selected, the elements it contains will not be part of another group.
- We will end up with 5 groups of 5 (example: the countries, there will be a European country group, another for the Middle East, etc.). All elements in a group are close to each other, so we have succeeded in uniting the identical or close elements within the same group as clustering.

- The final idea is to select the element(s) that represent the most of the group, so all the elements of the group are compared by their similarity with the group.
- Once all these tasks are done, if we want to select 5 elements (pairs) in input for the extraction of the new pairs, we will use the first represent of each group. Indeed, in our example the 25 pairs in input is transformed to 5 groups of 5 and ranked by their similarity to select them. Moreover, if we want 10, we take the first two elements that represent the most of each group.

Using a group representative improves the results in many ways. First it will produce a wider range of new pairs and since the input size is reduced the computation time is reduced too.

3.4. K-Means Selection

The last method presented is the K-Means Selection. For that we use the « k-means|| » implementation which is a parallelized variant included in MLLIB with Spark. The K-means algorithm is one of the most used clustering algorithms. It can be used for both regression or classification. One of the advantages is that it is unsupervised learning, so we can work with unlabelled data. The only argument is the value of **k** that represents the number of clusters we want.

This Figure 10 presents an overview of how the algorithm works. The example presented in the figure is in dimension 2 instead of 100 in our case. The first step of the K-means is the random generation, because it is random, the clusters obtained with the K-means can be different for different executions. However, a seed can be used to obtain the same result each time. In our case, to obtain 5 clusters we set **k** to 5, but all the clusters (groups) will not necessarily be of the same size contrary to the Pair Selection Methods (Euclidean and cosinus 3.3).

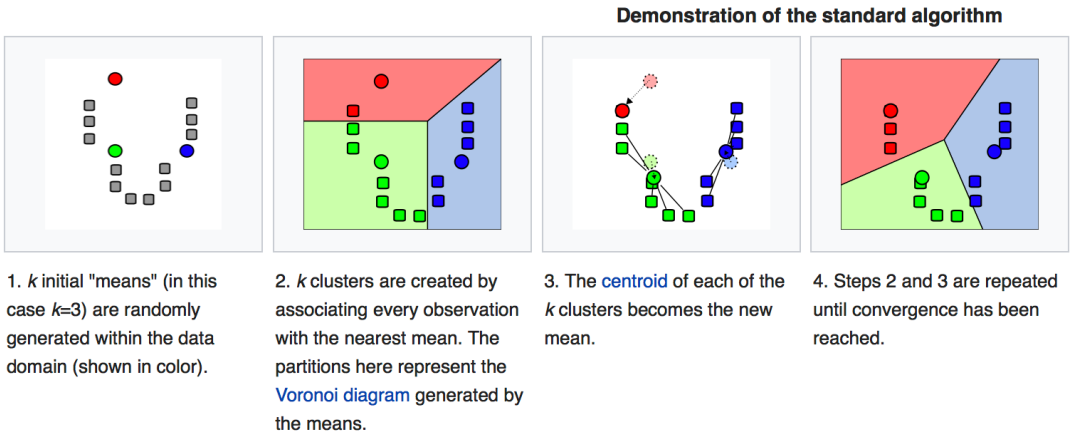

Figure 10: Short explanation of K-Means [18]

Our method uses the k-means algorithm to group pairs into clusters (what we call our groups). The next step is to compute for each group the k-means cost of each element, which is the **sum of the squared distances** of the points to their nearest center. The elements of each group are then sorted in an ascending order based on their k-means cost. An element with a low-cost value means that this element is close to the centroid of the group and therefore represent it well. Although with the K-means method the groups can be of different sizes, the method loop over all the groups each time picking (removing) the best representative until we obtain the desired number see Figure 11.

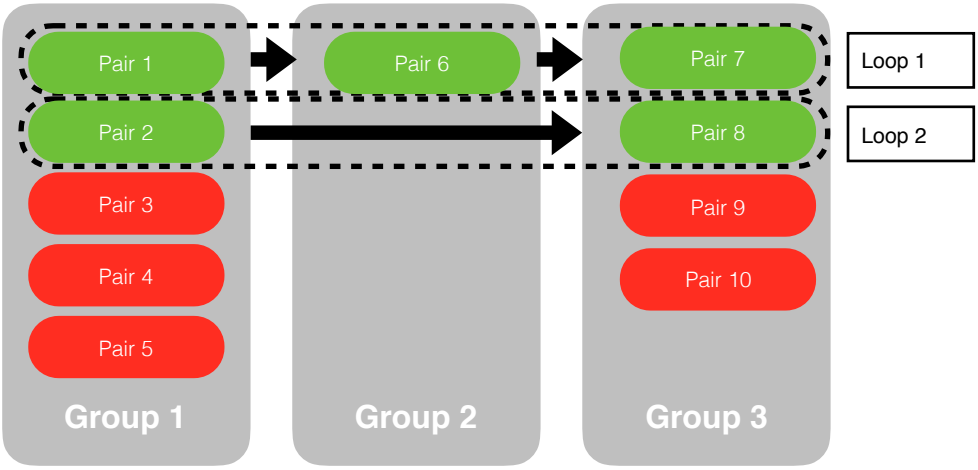

Figure 11: Schema of k-means pair selection with uniform groups, green means selected pairs returned and red no selected pairs.

The schema is an example of the k-means pair selection. There are 3 groups of different sizes for a total of 10 pairs. To select 5 pairs, the method loop over the groups each time selecting the first pair in the ranked order. After the first loop, only three pairs are selected therefore the loop is continuing selecting the remaining pairs until the required number has been selected.

4. Relation Extraction

4.1. Introduction

In this chapter, each step of the relation extraction process is explained. The code analyzed was originally developed by Matúš Pikuliak [19] and is working with Gensim in local mode (one machine). One the project contribution was to understand this code and to deploy it on Spark using RDD to improve the performance and the ability to handle more data. All these modifications needed to be performed without altering the results. Moreover the new code had to work with a model created by Mllib.

4.2. Explanation

The Figure 12 shows the UML class diagram of the refactored code. It gives a useful overview of the new code structure and each class will be briefly presented in the following chapter. There are 5 classes, one of them is "**PairSet**" which contains the list of the input Pair (semantic relation) from the input text file. The "**Pair** "class is used to create a pair of two embedding words. The "**Embedding**" class is used to create an embedding word, it contains the word and the vector representation of it (stored as a list of float). The classes "**Result**" and "**ResultList**" are only used to store the results (new generated pairs), **also named "candidate"**, and sort the results by score, ranked by similarity.

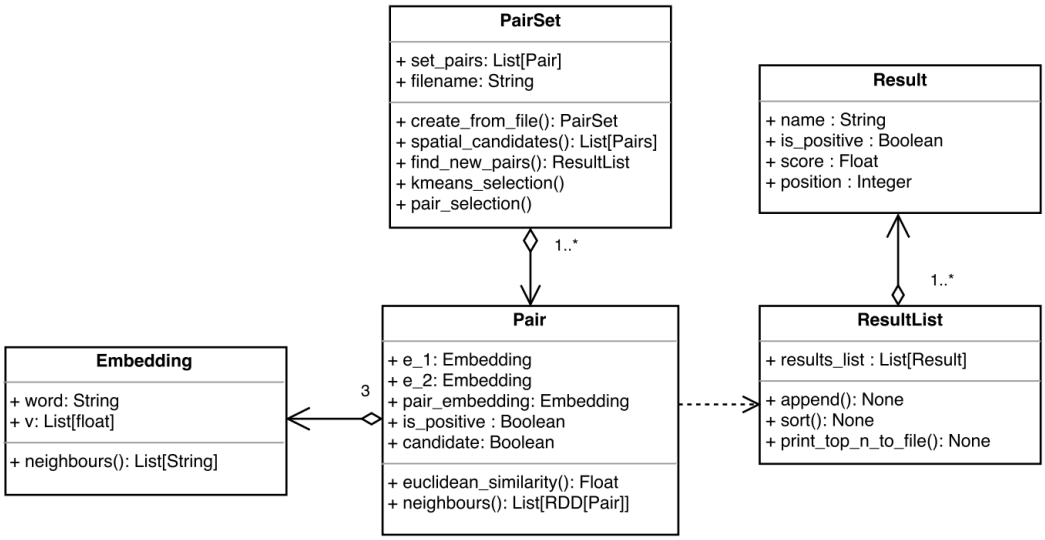

Figure 12: UML Class diagram

- Embedding: This class is used to create the structure of embedding words, so each embedding is a distinct word which contains one **word** and his vector representation named **v**. Its method "**neighbors**" return a list of the **N** closest words in the vector space. In the project context, we are interested in the 100 closest words.
- Pair: A Pair represents two words like « Paris » and « France ». It contains three embedding instances, one for each word of the pair and one for a new word like « Paris - France ». The third word is created using the first two embedding words by merging them and subtracting the vector of word1 with word2. Positive and candidates are boolean variables indicating status of the pair. At least one of them should be false. So, if a candidate is true, it means that there is a correct new candidate during the relations extractions. Positive says if it is input pair load by the text file or a generated pair. The **neighbors functions** call the similar function to embedding class (see to 4.4), and for **euclidean_similarity** please refer to 4.6 chapter.

- PairSet: This class is a set of Pairs, that contains a least 1 pair. The property "**set_pairs**" contains all the pairs given by the user in the text file in input. **Filename** contains only the name of the input file. This class is like the main one, because by this class we can launch some task like generates new pairs, pair selection.
- Result: This class is used to store the similarity results of one pair. **Name** contains the word of pair like « Paris - France », **is_positive** is to know if it's a pair from the input file. **Score** contain the similarity result between all the input pairs, in order to found the best candidates. **Position** is used at the end of the process in order to rank the best candidates.
- ResultList: This class is a list of results, which contains at least 1 **instance of Result**. The method "**Append**" is used to add new Result instance. The "**sort**" method is used to rank all the result elements by similarity score. The "**print_top_n_to_file**" method is used to choose the number of ranked candidates we want to print and save in the output text file.

4.3. Initialization

The role of this part is to retrieve new pairs of the same semantic relation type. The idea is to give as input a list of pairs of the same relation type (example: A is the capital of B, A is part of B). These input relations are used to retrieve new relations of the same type. The relations obtained depend on different factors such as the model, the semantic relation between the words, the corpus. Some relations have a stronger link than others which can be more complex and therefore harder to get. The chapter 4.4 contains more explanation about how it works. The script is run.py.

The first step (1 in the figure 13) is the initialization task, in which the user has to create a text file with one or more similar semantic relations. This task is represented in gray in the figure 13 and it is also processing all pairs to obtain **embedding** objects. An embedding object is containing a word and its word vector. For example, the pair « Paris - France » is split in two embedding objects for « Paris » and « France » and also an additional embedding object for the relation word « Paris - France » with a word vector equal to the difference between the vector coordinates of the two words. Finally, the program output a **PairSet** which contains a list of the most similar **Pairs** to those given in the input file.

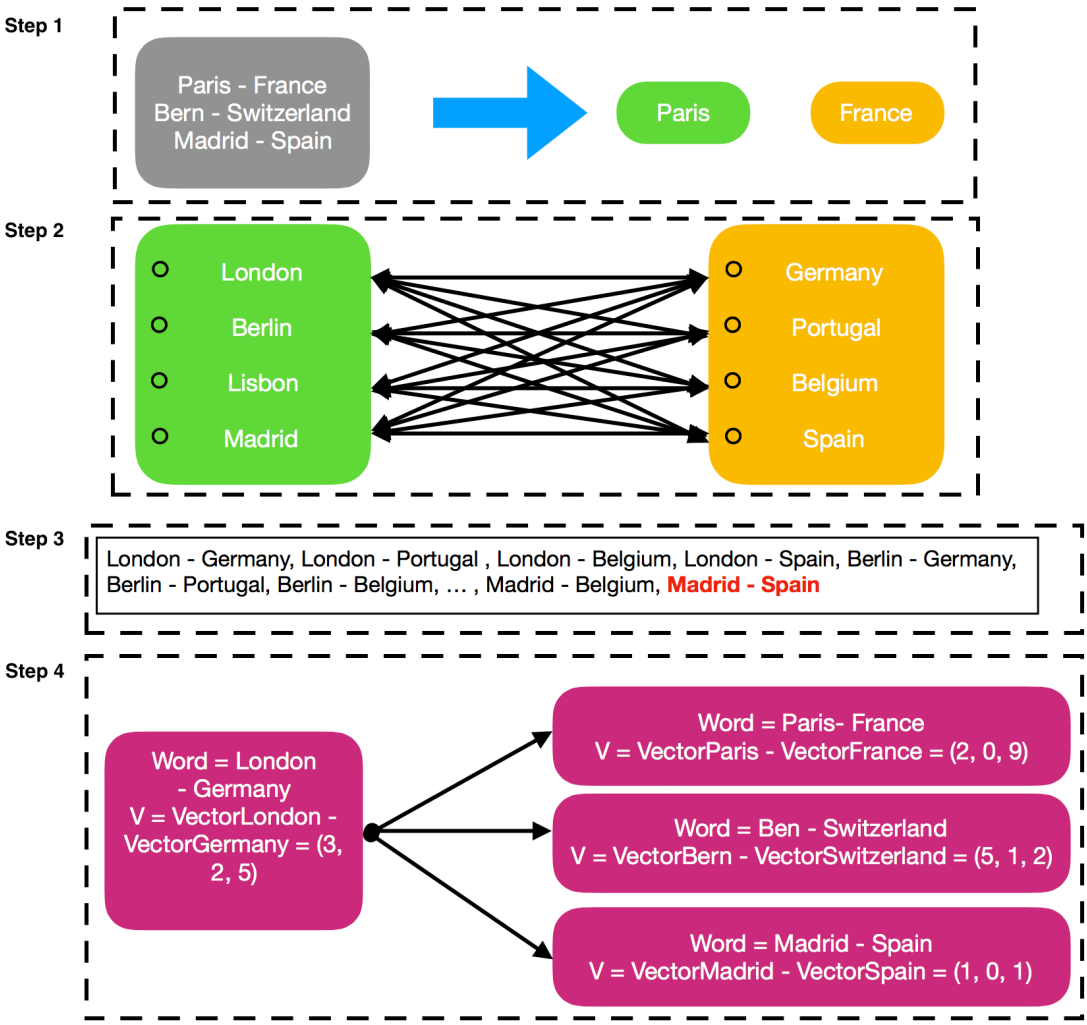

Figure 13: Schema of the main steps in our method for relation extraction

Here are described the input and output of each step. The figure 13 is a simplified view of the chapters 2.3 and 4. The idea of it is to give a quick overview on how the algorithm works.

- Step 1: The gray square represents the input text file which contains pairs given by the user. Each pair is transformed into embedding objects as described in the section 3.1.
- Step 2: In this step the input is the two embedding objects and the purpose of this step is to generate new candidates. The green and orange squares represent the **neighbors** of the embedding objects. Then a **Cartesian operation** is performed in order to obtain new pairs. The output of this step is a list of pairs generated by the Cartesian operation. In the figure 13, it is showing how the process is processing the first pair « Paris - France », and by retrieving the 4 closest neighbors pairs the Cartesian operation generates 16 new pairs.

- Step 3: This step receives as input the previously generated list of pairs and remove the duplicate and the pairs already present in the input file (for example: « Madrid - Spain » in the figure 13). This step returns a cleaned list of pairs.
- Step 4: The last step decides which pairs from the cleaned list of pairs are the best ones. Each pair from the output list is compared to all the input pairs using a **Euclidean similarity**. Since each pair is represented as an embedding object (containing the word vector of the pair), the word vector of each pair is used to compute the Euclidean similarity (the section 4.6 presents the detailed formula). At the end we obtain the list of the retrieved pairs ranked by the Euclidean similarity to the input. In the figure 13, on the left square it is the pair tested for the Euclidean similarity compared to the pairs from the input file on the right.

4.4. Generation of new pairs

The second step of the figure 13 is finding the most similar words (closest neighbors) to each of the pair component. The closest neighbors are the words whose word vectors are the closest to the input word vector. The green color square contains the words close to the pair component « Paris » and the orange the words close to the pair component « France ». Each group of neighbors is stored in a RDD and the **Cartesian product** between the two RDD produces new pairs.

```
def neighbours (self, sc, size=100):
    """
    Generates candidates for this given pair as product of neighborhood of its two embedding.
    : param sc: SparkContext
    :param size: Integer
    :return: list of Pairs in RDD type
    """
    ng_1 = sc. parallelize (self.e_1.neighbours (size))
    ng_2 = sc. parallelize (self.e_2.neighbours (size))
    rdd = ng_1.cartesian (ng_2). cache ()
    return rdd. flatMap (lambda x: [(x [0]. word+'-'+x [1]. word, Pair (x [0], x [1],
                                                candidate=True))] if x [0]. word != x [1]. word else [])
```

Figure 14: Script of Cartesian Product to generate new candidates

The Figure 14 shows the Spark code that generate the new pairs. The argument **Size** corresponds to the number of closest neighbors to retrieve for each pair component. **e_1** and **e_2** are the two embedding objects of the pair. Each group of neighbors is parallelized into a RDD in order to take advantage of the RDD operations. The function also output a RDD that is the result of the Cartesian operations between the two groups of neighbors. The last line is here to check that no pair is composed of the same word for the two components.

$$\text{similarity} = \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|_2 \|\mathbf{B}\|_2} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \sqrt{\sum_{i=1}^{n} B_i^2}}$$

The formula is the function that calculate the distance between two word vectors A and B. There is a function in MLLIB named *findSynonyms(w, n)* that return for the word **w** is the **n** closest neighbors. This function is called on a Word2Vec model to get the most similar words to one input word. This function use the cosine similarity [20] to find the nearest words and return them as a list of tuples that contain the word and its cosine distance.

4.5. Duplicate pairs

The third step of the figure 13, checks if there is no duplicate between the generated pairs and the pairs given as input and also between the pairs generated for each input pair. This step is basically removing all the pairs from the generated pair list that are also present in the input pair list. In our case, we set the number of closest neighbors to retrieve for each pair component to 100. So the Cartesian product produces 10,000 pairs (100 words * 100 words) for each input pair. Therefore there is high likelihood to find duplicates between the pairs generated for each input pair.

4.6. Pairs selection

The pair selection is the step that selects from the entire list of pairs generated in the previous step the best to use for our goal. To select these pairs the program computes the similarity score between each new pair and all the input pairs, as described in the step 4 from the Figure 13.

$$score(C, P) = \sum_{i=1}^{s} \frac{1}{s} \cdot \frac{1}{1 + \sqrt{\sum_{j=1}^{n} (Pij - Cj)^2}}$$

The *score* formula is used to measure the similarity between one pair and all the input pairs. This formula returns a similarity score between 0 and 1, the higher meaning that the pair is very similar. P represents all of the input pairs, C is the new pair to score, s is the number of elements in P. The idea behind this formula is to compute the Euclidean distance between a tested pair and an input pair, and multiply by one divided by the size of the input pairs set in order to have the same weight between them.

The final step is sorting the pairs by their similarity scores and returns the list of pairs with the highest similarity score. By default the program is returning 100 pairs but this can be modified if needed.

4.7. Visualization

In this part, we show an example of a 2D projection of the word vectors in a multi-vectorial space. In this projection only the words related to the relation "Capital - Country" are shown in order to render this visualization as readable as possible. This 2D representation gives just an idea of the word vectors location, but does not correspond to the real distribution. Indeed, in our case, each word vector is composed of 100 coordinates. Therefore we used the distributed stochastic neighbor embedding method [21] also named t-SNE in order to perform a dimensionality reduction and produce this scatter plot.

Figure 15: Visualization of embedding word for capital - country relation. The red circle contains the most of the time country/capital/city from South America, in blue circle North of Europe and in the Green circle from Asia.

In the Figure 15, the countries (from the relation « City - Country ») from the same continent are located close together (their context are likely to be similar). We chose this example because it is easy to interpret but this can also be seen with other relations. In the

top right of the figure 15 are located all the countries from the northern countries(blue circle). Similarly, there is a big cluster of Latin American countries in the top left part (red circle) of the figure. Our N-gram words are also correctly located. To see more detail of some part see the figure 16.

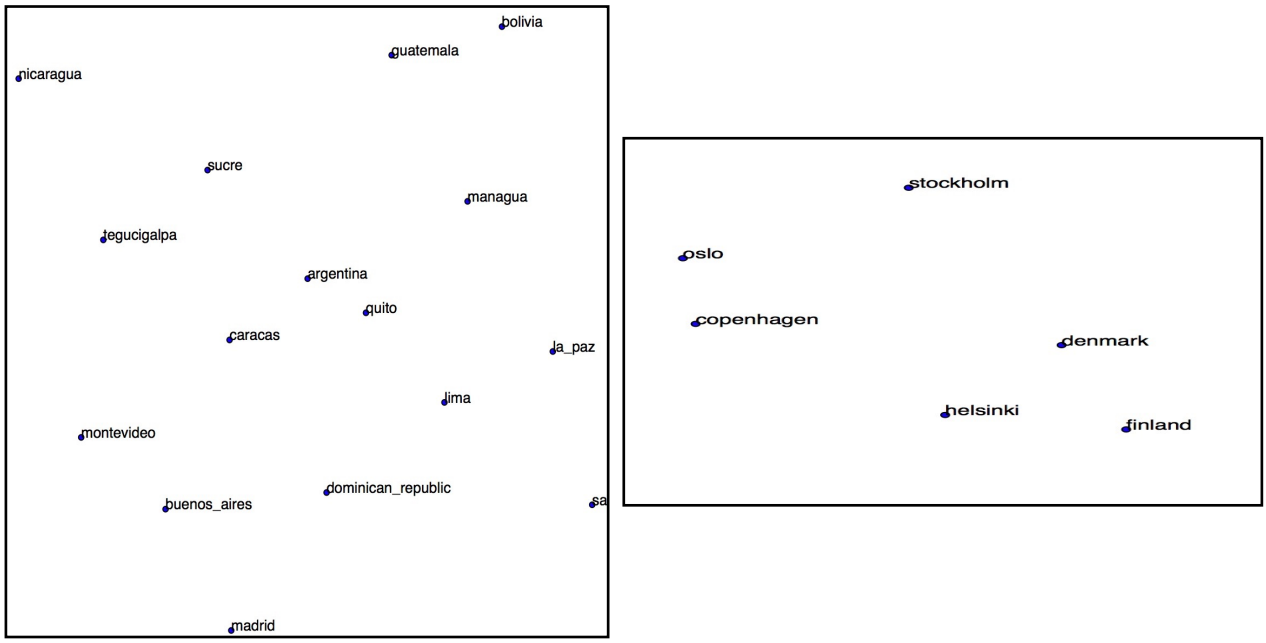

Figure 16: Zoom on some parts of the figure 15.

4.8. More explanations

This section gives more information on how to run the scripts. During our experiments, we wanted to only return 100 pairs in order to evaluate their semantic relation. But this number of returned pairs and the number of neighbors considered for each input pair can be changed. The arguments to set are:

- -o: This parameter is the output file where the results will be written.
- -t: This is the number of results to return.
- -d: This parameter is the number of neighbors considered when generating candidates.
- -s: This parameter is an optional boolean parameter used to activate the input pairs selection.

Example on how to run the script "run.py": *run.py -o output.txt -t 100 -d 100*

5. Results

This section is presenting the results obtained during the project. We used a model created with Spark with this set of parameters:

- Number of vector size 100,
- Windows for the context 10
- Learning rate 0.05.
- A corpus "enwiki" from wikipedia dump, which is corpus containing articles from various domains.

After the cleaning and training part, our corpus contains 7 673 265 words with one hundred coordinates for each word. The model used N-Gram and StopWord list in the pre-processing part. It is also possible to create a second model with a different configuration for preprocessing part. The table 1 shows that the model (Model 2) using the stop word list is considerably improving the execution times. Indeed, the corpus after pre-processing part is smaller while the number of distinct words was multiplied by almost 4 times. In our experiments we are essentially focused on the model 2.

Table 1: Configuration of Word2Vec Model

Configuration	Model 1	Model 2
N-Gram	Not used	Used
Stop Word List	Not used	Used
Vectors Size	100	100
Learning rate	0,05	0,05
Num of Partions	4	4
Windows Size	10	10
Seed	42	42
Min Count	5	5
Execution Time	406m29.355s	288m40.239
Num of Distinct Word	2036173	7673265

5.1. Configuration

The tests were conducted using a local machine (iMac) and a cluster of computers from the University of Fribourg. The local machine was used to locally compute the original code and the new code with spark. The cluster was only used to run the new code with Spark distributed mode. The table 2 is the detailed information about the machines used.

Table 2: Configuration of the machines

Configurations	iMac	Cluster
RAM	24Go	128Gb
Node(s)	0	50
CPU	4	32
CPU Name	3,3 GHz Intel Core i5	Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10GHz

5.2. Results of the relations extraction

Each presented method is, given a pair list, outputting a new list of pairs. In order to measure the quality of the pair lists generated by the presented methods, we measured the nDCG score and the precision of these lists. Indeed, to measure the precision of a pair list, we count the number of correct pair over the total number of pairs. A pair is considered correct if it is representing a semantically true relation between the two components. The nDCG score is used in order to take into account not only the precision of the pair list but also the position of the correct pair into the list. We calculated these scores on the results obtained by providing five different types of semantic relations in entry. For each type of relation we provided a list of 25 pairs and, each method selected from this list the number of pairs to effectively use.

5.2.1. Relation City - Country

The relation between cities and countries is often easily identifiable because the two words are likely to share the same context. This is a relation easy to validate, therefore we expected to obtain good pairs in output. In the table 3, we present the results obtained for S = 100 pairs in output. A precision of 0.01 means that the output contains 1 true candidate for 100 returned.

Table 3: Measures of the relation City - Country, S is the amount of the value return for the evaluation part and N corresponds to the number of neighbors set during the relation extraction part.

N = 100 S = 100	Word Count Selection		Cosine Selection		Euclidean Selection		K-means	
City - Country	Precision	nDCG	Precision	nDCG	Precision	nDCG	Precision	nDCG
5 Pairs	0,02	0,70881107	0,1	0,91167856	0,13	0,88684916	0,14	0,82661220
10 Pairs	0,05	0,60532540	0,2	0,73101152	0,17	0,82316525	0,19	0,76836830
15 Pairs	0,13	0,83873352	0,22	0,75844742	0,18	0,77856340	0,19	0,83177022
20 Pairs	0,16	0,80463778	0,2	0,82103856	0,19	0,86901169	0,18	0,80714244

The best score is obtained when one the input pair list is composed of 15 Peers. In general for this relation, the precision and the nDCG score is very high. The nDCG is particularly high, which proves that this relation has a very good score for information retrieval; which corresponds to our accuracy of the extracted relations. Moreover, we observed that we obtain a good precision and nDCG scores using our methods except for the « Word Count Selection ». All the other methods give good scores with 10 pairs in entry and there is a small score variation with a greater number of pairs. The fact that our methods provide good scores with fewer pairs in input shows the improvement realized.

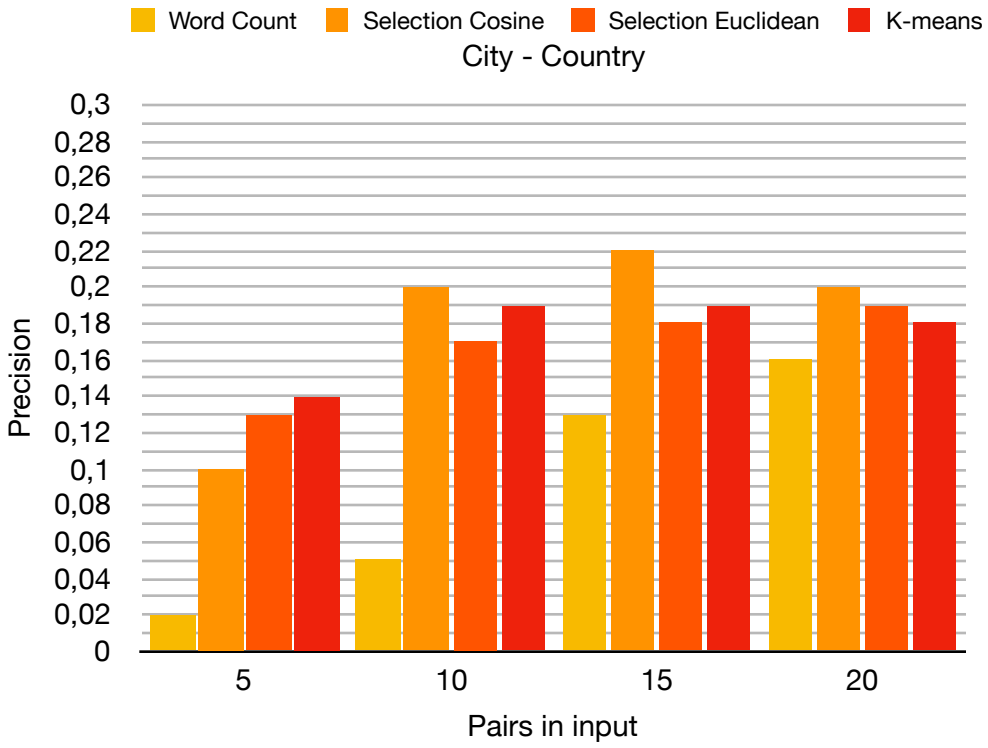

Figure 17: Precision by the number of pairs in input for City - country

In the figure 17, we present, using a bar chart, a visualization of the precision obtained using each method. We obtain a lower precision score when using the « word count selection » algorithm compared to the other algorithms.

Table 4: Excerpts of the Input Text File that contains the input pair

Input Text File
athens greece
baghdad irap
bangkok thailand
beijing china
berlin germany

The table 4 gives an example of an input file for the relation « City-Country ».

Excerpt from the output file
1 damascus - syria Damascus - Syria : Damascus - Syria
1 istanbul - turkey Istanbul - Turkey : Istanbul - Turkey
1 kabul - afghanistan Kabul - Afghanistan : Kabul - Afghanistan
1 algiers - algeria Algiers - Algeria : Algiers - Algeria
1 budapest - hungary Budapest - Hungary : Budapest - Hungary
1 bogota - colombia Bogota - Colombia : Bogota - Colombia
1 tehran - iran Tehran - Iran : Tehran - Iran
1 auckland - zealand Auckland - Zealand : Auckland - New Zealand
1 beirut - lebanon Beirut - Lebanon : Beirut - Lebanon
0 baghdad - syria
0 melbourne - adelaide
1 montevideo - uruguay Montevideo - Uruguay : Montevideo - Uruguay
0 colombia - ecuador

Table 5: Excerpt from the output file of candidates after evaluation

The table 5 shows a sample of the pairs output after the evaluation part, for 5 pairs in input. The first value (0,1) defines if a pair is considered true or false. When a relation is true there is a second part which is the result from our Knowledge Base.

Example:

« damascus - syria » is the first part which is extracted by our algorithm and « Damascus - Syria: Damascus - Syria » is the second part of Wikidata(KB). The Wikidata part is used to check if the first part is true.

5.2.2. Relation Capital - Country

Capital and countries words are often close together inside a text and they also share the same context. Like for the city - country relation, the same trend is observed for ten pairs in input. Moreover we can notice the good performance of the K-means selection method with only five input pairs.

N = 100 S = 100	Word Count Selection		Cosine Selection		Euclidean Selection		K-means	
Capital - Country	Precision	nDCG	Precision	nDCG	Precision	nDCG	Precision	nDCG
5 Pairs	0,02	0,70881107	0,06	0,95079412	0,04	0,67415223	0,14	0,82661220
10 Pairs	0,04	0,62434582	0,15	0,71701085	0,12	0,77378146	0,14	0,72377020
15 Pairs	0,1	0,85701445	0,15	0,71602478	0,12	0,71252166	0,16	0,80219797
20 Pairs	0,12	0,81090997	0,13	0,80935219	0,13	0,86070302	0,13	0,77761524

Table 6: Measures of the relation Capital - Country

As we can see in the diagram (figure 18), the precision score of the « K-Means » method for 5 pairs in input is 6 times higher than with the « word count » method. However, this difference shows the importance of correctly selecting the pairs of the same relation.

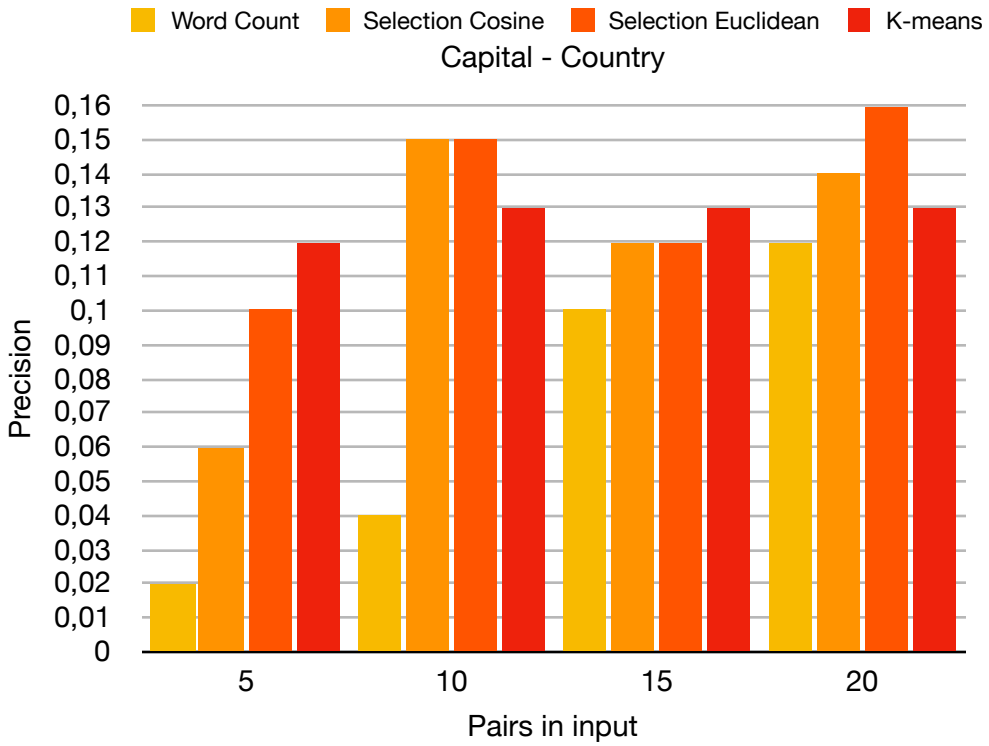

Figure 18: Precision by the number of pairs in input for Capital - Country

5.2.3. Relation Last Name of US Politicians - Place of birth

This relation is a more complex relation compared to the "city - country" one, the link between the two words of the relation is not really straightforward. Usually when there is an article or a paper about a politician, the context is not containing its place of birth. We can then expect that a more complex relation will give us less accurate results.

N = 100 S = 100	Word Count Selection		Cosine Selection		Euclidean Selection		K-means Selection	
Politicians - Place of Birth	Precision	nDCG	Precision	nDCG	Precision	nDCG	Precision	nDCG
5 Pairs	0,24	0,5552206	0,2	0,4786833	0,15	0,4372689	0,13	0,4454016
10 Pairs	0,14	0,4181354	0,08	0,3462496	0,12	0,4171944	0,04	0,2613305
15 Pairs	0,14	0,4406469	0,07	0,3268377	0,08	0,3401740	0,05	0,2750577
20 Pairs	0,15	0,5452581	0,14	0,4936471	0,12	0,5152100	0,13	0,4637468

Table 7: Measures of the relation Last Name of US Politicians - Place of birth

This can be explained by the fact that a complex relation contains more randomness. Indeed, during the candidates generation part, if the relation is not straightforward, the elements are more distant in the multi-vectorial space and the results are more various. In addition, the nDCG score is low, which means that the extracted relations are less precise and thus produces more mistakes.

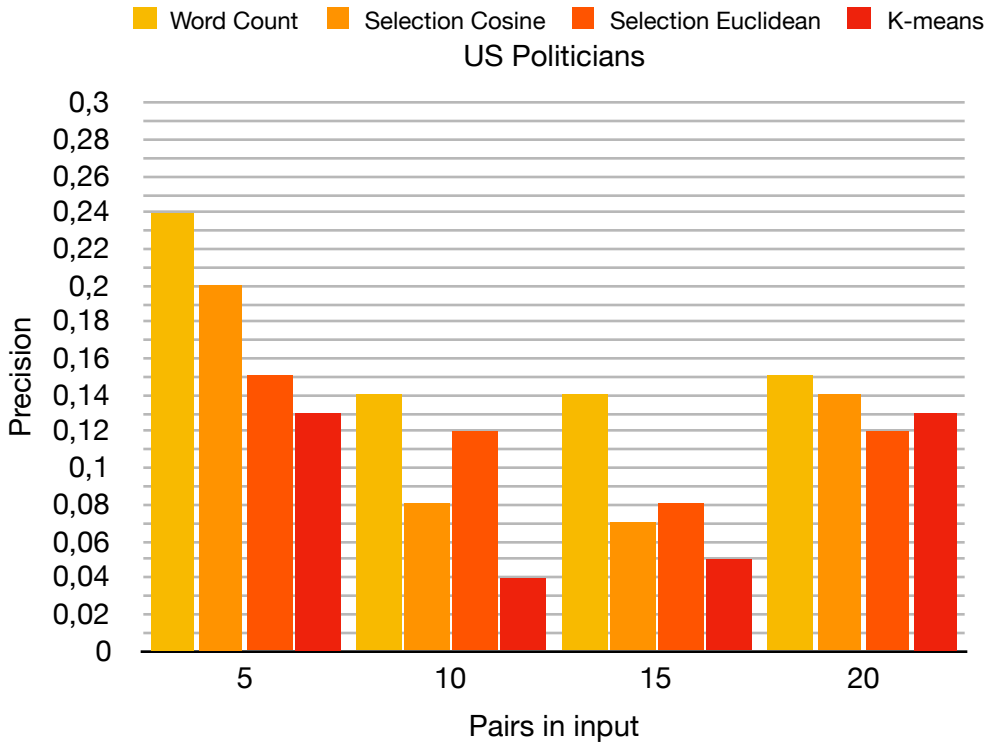

Figure 19: Precision by the number of pairs in input for US Politicians

5.2.4. Relation Masculine Words - Feminine words

The relation between masculine and feminine words is an interesting example for the relation extraction. Therefore word gender relations have been evaluated (e.g.: « brother - sister » or « policeman - policewoman »).

N = 100 S = 100	Word Count Selection		Cosine Selection		Euclidean Selection		K-means Selection	
Genre	Precision	nDCG	Precision	nDCG	Precision	nDCG	Precision	nDCG
5 Pairs	0,09	0,66652655	0,08	0,80427065	0,09	0,60070316	0,08	0,68337734
10 Pairs	0,09	0,63603037	0,08	0,67698804	0,09	0,62002733	0,11	0,71923196
15 Pairs	0,09	0,65300969	0,08	0,64288351	0,09	0,64938132	0,08	0,66369696
20 Pairs	0,08	0,64283234	0,06	0,64838506	0,09	0,64825878	0,08	0,64547243

Table 8: Measures of the relation Masculine Words - Feminine words

This type of relation is more subjective so there is not really any knowledge base that can gauge whether it's true, false or just half true/false. The notation will therefore not be binary, but between 0 and 2 with 1 for a half-true. This notation influence the nDCG score for a more accurate measurement. With this relation, we obtain almost the same result for all the methods.

Examples :

- waitress - waiter: This relation is good so it's equal to 2.
- waitress - bartender: This relation is hard to evaluate because "bartender" is a gender-neutral term. So we can evaluate this relation to half-true and the score is equal to 1.

5.2.5. Relation Nationality

The relation between nationality and country words is the last tested example for the relation extraction. We obtain the best results with this relation. We can notice the good performance for the « Euclidean Selection » with 5 pairs, 31 true candidates for 100 of candidates retrieved. The high nDCG score means that the most of true relations are in the ranked high in the list of the candidates retrieved.

N = 100 S = 100	Word Count Selection		Cosine Selection		Euclidean Selection		K-means Selection	
Nationality	Precision	nDCG	Precision	nDCG	Precision	nDCG	Precision	nDCG
5 Pairs	0,05	0,81407443	0,25	0,90435066	0,31	0,94762912	0,15	0,82829639
10 Pairs	0,16	0,86269952	0,28	0,91776613	0,31	0,94762912	0,19	0,90230856
15 Pairs	0,22	0,91805851	0,34	0,93214792	0,3	0,93260822	0,27	0,92474019
20 Pairs	0,38	0,95105496	0,36	0,93158347	0,39	0,94548011	0,36	0,94501216

Table 9: Measure of the relation Nationality

In this Figure 20, we can see the global good performance for our method expect for « word count selection ». The Cosine and Euclidean methods start with high precision scores.

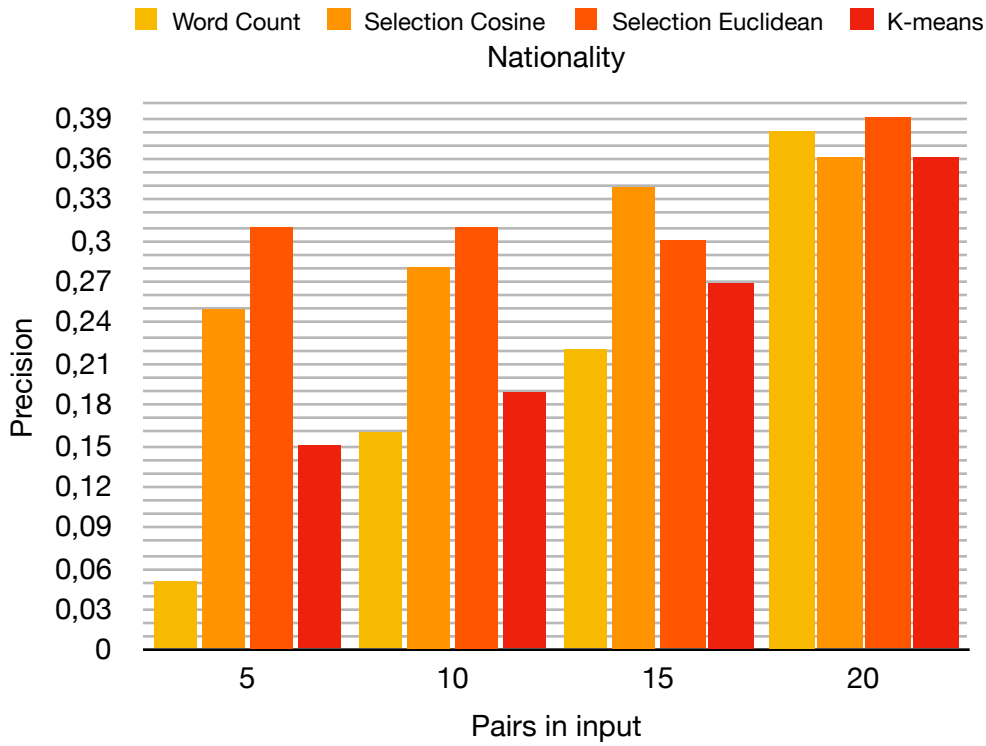

Figure 20: Precision by the number of pairs in input for Nationality

5.3. Impact of neighbors

In this part, we chose to apply the K-means selection method on the Nationality relation. We chose this relation because this is the one with which we obtained the best results, so we can analyze the variations in them. The idea is to measure the impact of increasing the number of neighbors. We only used one relation with one method, because we obtain almost the same result with the other example. For the measure we set the number of pairs returned (**S**) to 100, and we modify the value of N.

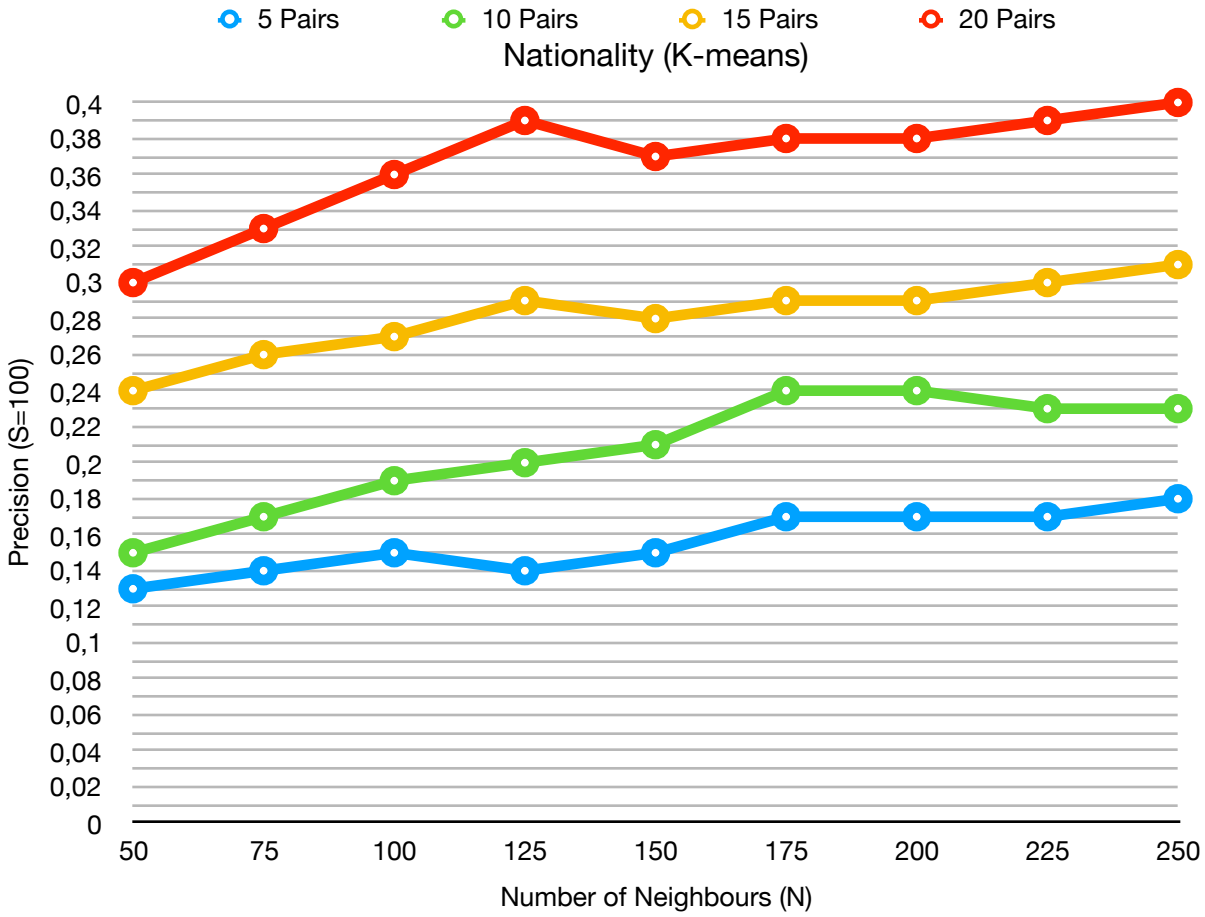

Figure 21: Impact of neighbors with K-Means selection for Nationality relation.

In the figure 21, we see a good precision with the value of N between 100 and 150. After this, the results tend to stabilize. It should be remembered that, for each pair in input, the method is generating N * N candidates. Moreover, increasing the number of neighbors can dramatically increase the execution time. There is no single solution, it depends on many factors such as the size of the corpus, the number of neural network used, the number of words, so we must select a number N that produces good results without increasing too much the execution time. In our experiments, an N value of 100 is a good solution.

5.4. Impact of increasing the number of returned pairs

The objective of this part is to measure the impact of increasing the number of pairs returned. In effect, after the generation of the new pairs, all the methods use the Euclidean similarity to score each pair. They are then ranked in order to select the best generated pairs. If the algorithm works well, by reducing the number of pairs returned, the precision score will increase.

For this experiment, we chose the nationality relation and we define the number of neighbors to 200. We observe that we obtain a better result with N=100 and S=100 in 5.2.5 but there is not a huge difference between them. In the table 10, we decided to change the value of S to 50, 100, 150, 200. As we expected we obtain a higher precision for 50 pairs returned instead of 200.

	N = 200 S = 200	Word Count Selection		Cosine Selection		Euclidean Selection	
	Nationality	Precision	nDCG	Precision	nDCG	Precision	nDCG
S = 50	5 Pairs	0,06	0,870593493	0,44	0,945673107	0,54	0,965785866
S = 100		0,04	0,790584089	0,27	0,924672279	0,33	0,946993314
S = 150		0,02	0,790584089	0,21	0,901904203	0,22	0,946993314
S = 200		0,02	0,790584089	0,16	0,897566950	0,17	0,936654394
S = 50	10 Pairs	0,26	0,857442172	0,48	0,945716462	0,54	0,965785866
S = 100		0,16	0,830444692	0,29	0,927137189	0,33	0,946993314
S = 150		0,12	0,814863387	0,21	0,915201635	0,22	0,946993314
S = 200		0,09	0,806521200	0,17	0,906576715	0,17	0,936654394
S = 50	15 Pairs	0,36	0,950840386	0,58	0,958610012	0,5	0,959559039
S = 100		0,23	0,915902949	0,35	0,943590873	0,3	0,940492577
S = 150		0,17	0,898549643	0,26	0,927771900	0,22	0,927277277
S = 200		0,13	0,898549643	0,21	0,921159262	0,18	0,910668826
S = 50	20 Pairs	0,62	0,975699571	0,54	0,967745579	0,62	0,967758605
S = 100		0,39	0,957191960	0,38	0,939364159	0,38	0,952649902
S = 150		0,29	0,942622737	0,27	0,931112790	0,28	0,941363537
S = 200		0,24	0,931737817	0,22	0,919061693	0,23	0,927242282

Table 10: Impact of the number of pairs returned for the nationality relation

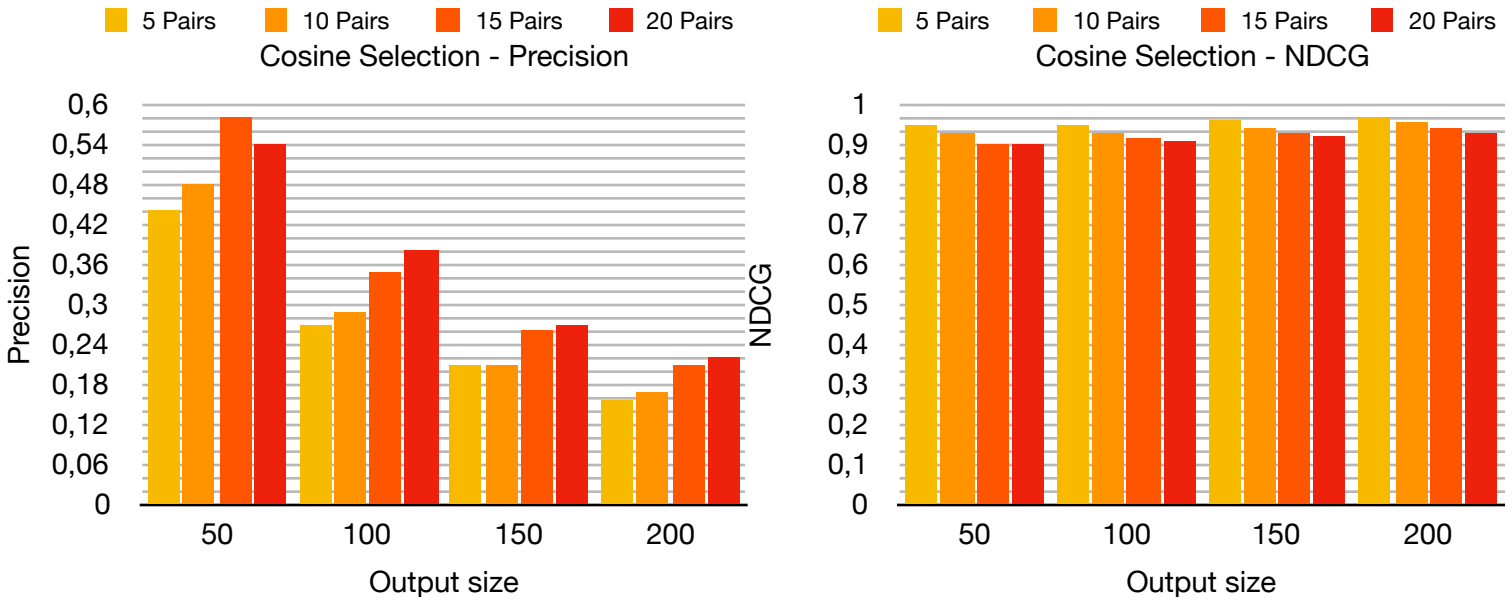

Figure 22: Precision and nDCG to measure impact of the output size

In the figure 22, the bar charts show the cosine selection, as we can see in the precision chart, the four methods produce almost the same « shape » with lower scores when the number of returned pairs increases. For S equal to 100 and 20 input pairs we obtain 38 true candidates, but for S equal to 200 and 20 input pairs equal we obtain 44 true candidates (a precision of 0.22). A smaller precision doesn't mean less true candidates. We observe that by increasing the returned pairs from 100 (27 for 5 pairs) to 200 (300 for 5 pairs) we obtain only 3 true new candidates. So, we can still get good candidates by returning more pairs but the likelihood of getting good pairs if reduced. The second chart is proving that increasing the size of the output list does not affect the nDCG score because the true candidates are still ranked in the top.

5.5. Results comparison

In this section we tested the 4 selection methods with all the relation types in order to compare them. The configuration is **S** and **N** equal to 100.

5.5.1. Precision

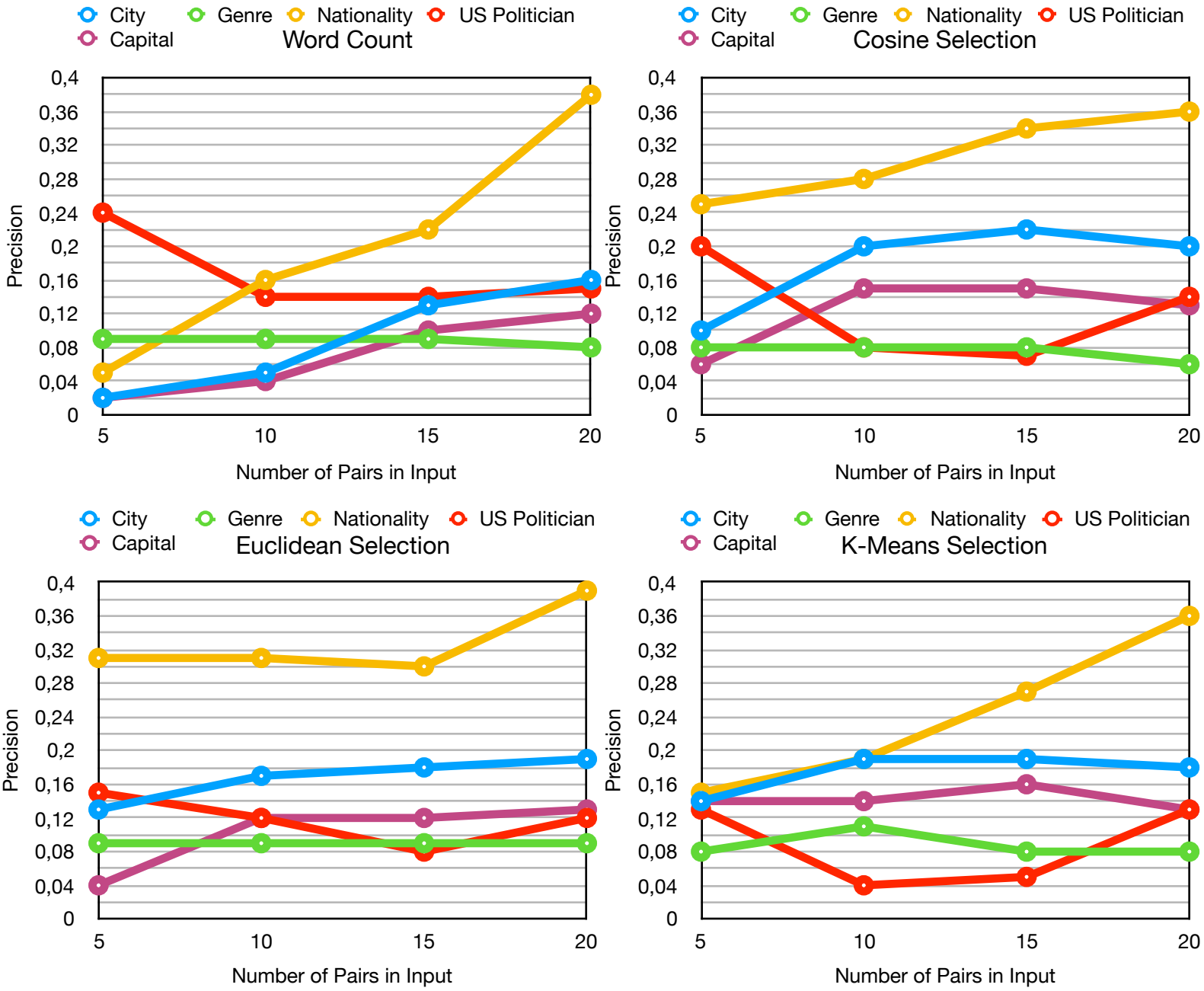

Figure 23: Precision between our methods with all the relations

In this part, we compared the precision between our methods with all relations. One interesting thing with Euclidean and K-Means Selection is that, we obtain almost the same results of 5 to 20 pairs, the curves are almost stable except for the nationality relation. For Cosine selection is almost the same but between 10 and 20 pairs. This finding is very important because it means that we get almost the same result with fewer input pairs, so we

got the expected results. Moreover, it can be interpreted that the simplest and most direct relations have better results.

Another interesting thing is the precision for 5 pairs, as we can see for K-means we obtain 4 relations with score equal or higher to 0.10, and 3 relations for Euclidean & Cosine and only one for word count selection. To conclude, the selection of pairs in input has huge impact in the precision, and with our algorithms we can propose a new method to improve it and reduce the number of candidates.

5.5.2. Normalized discounted cumulative gain

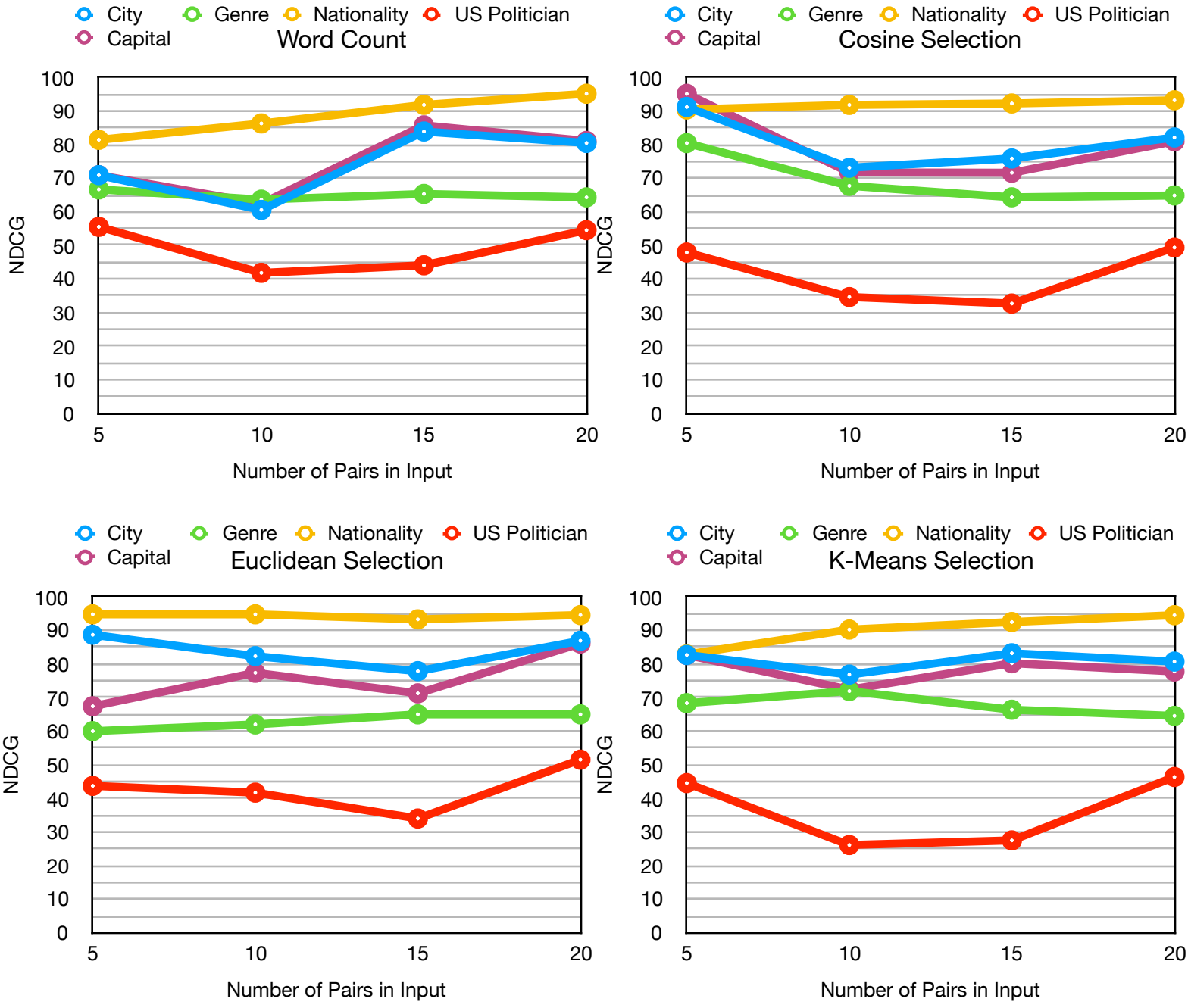

Figure 24: nDCG between our methods with all the relations

Across all the charts in the figure, we notice that the best nDGC score is comprised between 5 and 10. We see that it stabilizes at 5 pairs except for cosine we obtain the best result at 5 pairs. Moreover, for direct relations like « city » the value is stable, regardless of the number of peers in input. However, for a little more complex relation like « US Politician » the nDCG score has more variations.

5.5.3. Comparison of models

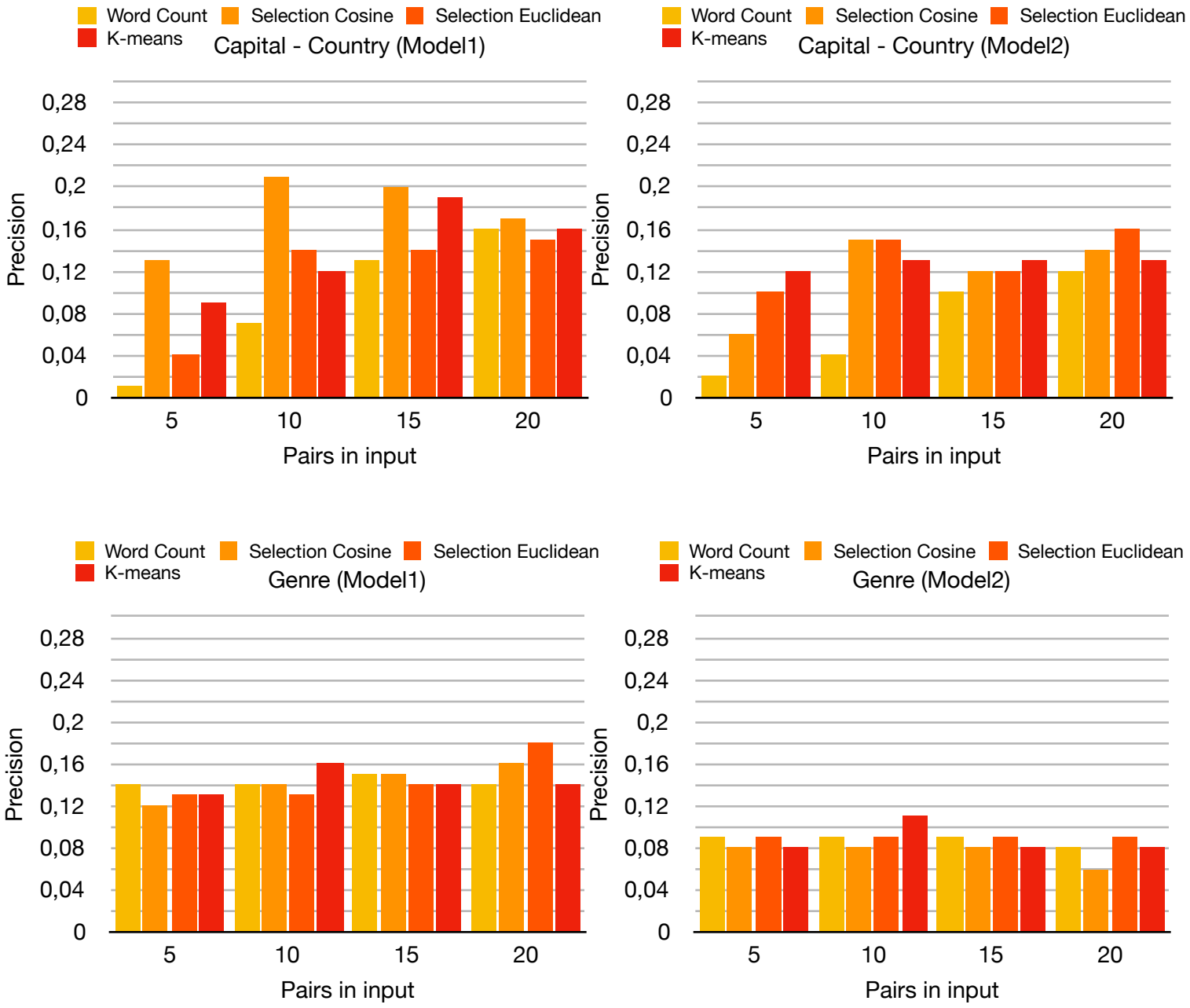

Figure 25: Comparison of models with precision

In this part we compared the models described in the section 5. But in our experiment we only used the model 2. The main difference between both models is that the model 1 doesn't handle n-gram whereas the model 2 do. Because of that, the model 2 contains almost 4 times more unique words than the model 1. These charts show the precision scores between these two with S and N set to 100. We obtain better results using the model 1, we guessed it is because this model contains less word. Moreover, in our experiments we never obtained a n-gram candidate. However, if the user wants to give « n-gram pairs » in input, it's possible.

5.6. Spark & Gensim

This part presents the comparison of the two frameworks Spark and Gensim. To fulfill this task, we created a model with the Mllib library of Spark (more precisely the model 1). Then we used our script to transform the model created under Spark into a Gensim compatible model. With all this process, it is now possible to compare whether there are any differences on the results.

	Mllib Spark		Gensim	
City - Country	Precision	nDCG	Precision	nDCG
5 Pairs	0,15	0,923527524	0,15	0,923135602
10 Pairs	0,23	0,916203658	0,23	0,916203658
15 Pairs	0,28	0,938486647	0,28	0,938528403
20 Pairs	0,29	0,932181575	0,29	0,932181575
25 Pairs	0,27	0,906023854	0,27	0,906023854

Table 11: Comparison Between Spark & Gensim

We can say that the results are almost identical even if there is a very small difference for the nDCG value. This difference is explained by the fact that even if the values are identical in the best ranked values, there are sensitive variations for the lowest ranked values. Then a good relation in the end of rankings can have an offset of 1 rank between them which causes this very small gap.

5.7. Execution time

To measure the execution time, we have run the original code on a single (local) machine (see configurations in 5.1). Then we also run our code on Spark using the cluster mode. We have also created a Spark version that run on a single machine but this one has not been used for the tests. The configuration used to obtain the best execution times with the cluster is:

```
> time spark-submit --py-files class_container.py --conf spark.driver.maxResultSize=4G --master yarn --deploy-mode cluster --driver-memory 42G --executor-memory 35G --num-executors 84 --executor-cores 4 run.py relations2/relations2/Genre25.txt -o hdfs:///user/jeremy/relations2/Genre25
```

- *--py-files class_container.py*: Allows us to load our python file
- *--conf spark.driver.maxResultSize=4G*: Mandatory for our program to avoid the error: *« Total size of serialized results of X tasks (X KB) is bigger than spark.driver.maxResultSize »*
- *--master yarn --deploy-mode cluster --driver-memory 42G --executor-memory 35G*: Use to configure YARN. In addition, the values of driver memory and executor memory can be decreased.
- *--num-executors 84 --executor-cores 4*: These items are optional, but with this configuration you can save a few minutes. Executor-cores is the number of concurrent tasks an executor can run, and num-executors is used to control how many executors will be allocated on the cluster.
- *run.py*: It is our main file to launch our script
- relations2/relations2/Genre25.txt -o hdfs:///user/jeremy/relations2/Genre25: These are the arguments of our script, the first argument is the path of the input file that contains the pairs (stored in HDFS). The second argument is the path where the output file will be saved, in our case in a directory of HDFS.

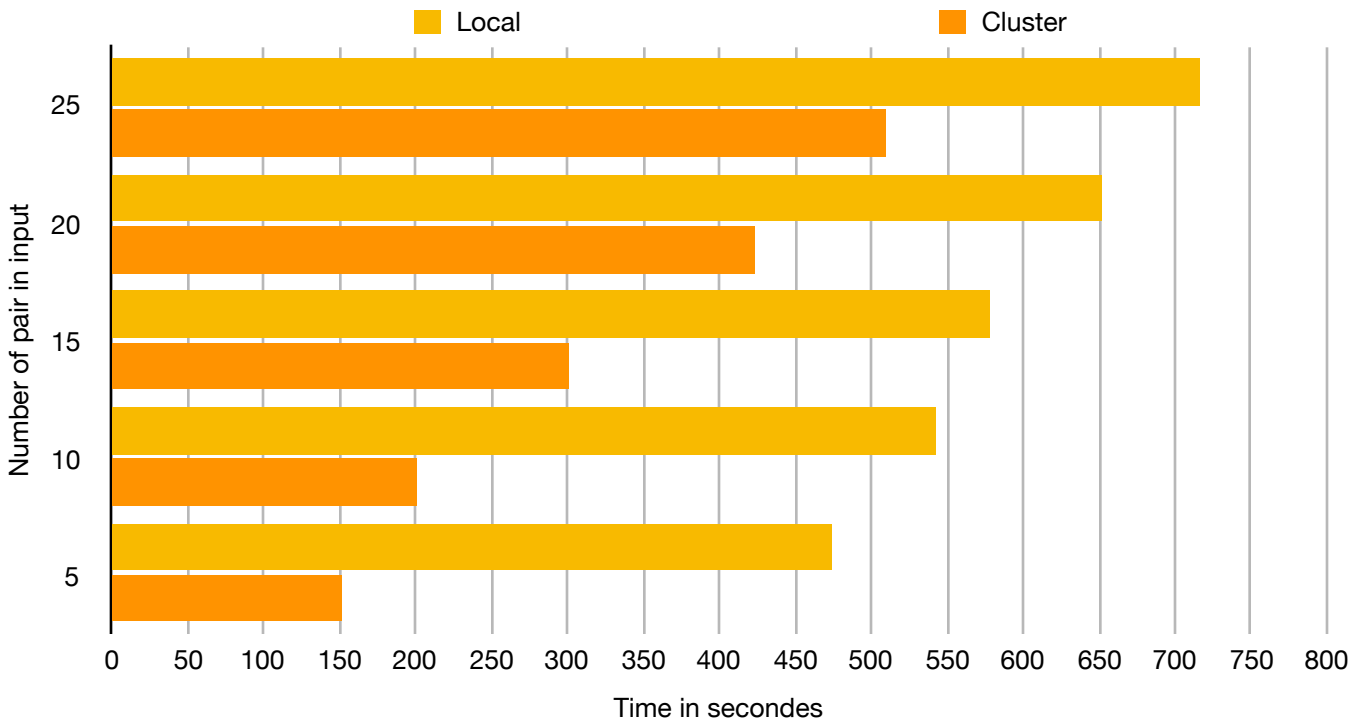

Figure 26: Execution time

More details are provided in the code and the readme of the program. They also contain information about optional arguments for the program.

The tests presented in the bar chart (figure 26) were conducted using with the same model that was used for the previous charts. This model was transformed to be compatible to work in local with Gensim. The cluster version bars are the execution time obtained by our program running on a cluster.

The figure 24 shows the best execution time, but it can exist a variation of 10%. However, there is no doubt about the execution time improvement offered by our Spark implementation. In effect Spark is especially powerful to load a model that is stored in HDFS.

Using fewer pairs in input (for example 5 pairs instead of 20) can considerably reduce the execution time. Therefore, if the results are (almost) identical it is better to favor the input with the lowest number of pairs. In our case, we could divide the execution time by almost 3 with the pair selection method improvement (for 25 pairs in input, there is 500 seconds instead of 150 for 5 pairs).

6. Conclusion

During this thesis we have modified a relation extraction algorithm in order to deploy it on Spark. With this operation the computation power was increased and thus the execution time was reduced. It is now possible to quickly create a word embedding model with Spark and using even bigger text corpus.

We introduced methods for selecting the best relation pairs to use as input in order to improve the precision of the results and decrease the execution time. Using these methods, has effectively improved the relation extraction process. Moreover, we have created a method to automatically evaluate the extracted pairs by using the knowledge base Wikidata.

This thesis enabled us to explain the essential tasks of the extraction relation from the processing of a raw corpus to the evaluation of the results. Regarding the measures, it is important to note that each step of the process can greatly influence the results. In the same way, it is important to correctly choose the corpus to use and also to properly process it.

7. Future Work

The expansion of the semantic Web could allow a better evaluation process, but also allow to measure if a relation is simple or complicate to identify. In addition, for increasing the performance even more, it might be necessary to use the Data-Frame structure instead of the RDD one in Spark. It might also be interesting to test with even larger corpuses. One improvement can be to link the extracted information to the knowledge base of the type of relation, before the generation of similar words in the relation extraction part. In order to detect if the generated words are true before doing the Cartesian product and the evaluation part.

Another thing is to try the « pairs extraction » algorithms and the « input pair selection method » with other words embedding algorithms like GloVe from the Stanford NLP Group.

8. References

[1] Wikipedia. Apache Spark. URL : https://en.wikipedia.org/wiki/Apache_Spark

[2] Apache Spark. Resilient Distributed Dataset RDD Documentation. URL: https://spark.apache.org/docs/1.6.2/api/java/org/apache/spark/rdd/RDD.html

[3] Wikipedia. Fault tolerance. URL : https://en.wikipedia.org/wiki/Fault_tolerance

[4] Le monde informatique. Figure. URL: http://www.lemondeinformatique.fr/actualites/lire-3-conseils-pour-eviter-de-transformer-un-data-lake-en-marecage-de-donnees-68482.html

[5] Wikipedia. Word2Vec. URL: https://en.wikipedia.org/wiki/Word2vec

[6] Mikolov et al. Efficient Estimation of Word Representations in Vector Space. URL: https://arxiv.org/pdf/1301.3781v3.pdf)

[7] Chris McCormick. Explanation about Skip Gram algorithm. URL: http://mccormickml.com/2016/04/19/word2vec-tutorial-the-skip-gram-model/

[8] Standford Edu. Softmax Regression. URL http://ufldl.stanford.edu/tutorial/supervised/SoftmaxRegression/

[9] Adrian Colyer. The amazing power of word vectors. URL: https://blog.acolyer.org/2016/04/21/the-amazing-power-of-word-vectors/

[10] Radim Řehůřek. Gensim Framework. URL : https://radimrehurek.com/gensim/

[11] Apache Spark. Library MLLIB. URL: https://spark.apache.org/docs/latest/mllib-guide.html

[12] Tomas Mikolov et al. Distributed Representations of Words and Phrases and their Compositionality. URL: https://arxiv.org/pdf/1310.4546.pdf

[13] Wikidata. About Wikidata. URL: https://www.wikidata.org/wiki/Wikidata:Main_Page

[14] Wikidata. Wikidata Query Service. URL: https://query.wikidata.org/

[15] Jeremy Serre. My bitbucket code reposit. URL: https://bitbucket.org/serrej/master_thesis/overview

[16] Wikimedia. Corpus Extracted. URL: https://dumps.wikimedia.org/backup-index.html

[17] Quora. Answers to visualization of a Word2Vec model. URL: https://www.quora.com/How-do-I-visualise-word2vec-word-vectors

[18] Wikipedia. K-Means. URL https://en.wikipedia.org/wiki/K-means_clustering

[19] Matúš Pikuliak. Original Code. URL https://github.com/matus-pikuliak/word-embeddings

[20] Wikipedia. Cosine similarity. URL https://en.wikipedia.org/wiki/Cosine_similarity

[21] Quora. Embedding visualisation t-SNE. URL : https://www.quora.com/How-do-I-visualise-word2vec-word-vectors