Progetto Ingegneria Informatica (5 CFU)
2017/2018

Riccardo Poiani, Mattia Tibaldi, Tang-Tang Zhou
Politecnico di Milano
Mercurio

June 4, 2018

Contents

References **2**

1 Introduction **3**

2 Web scraping **4**
2.1 Saving the data 5
2.2 Static websites 5
2.3 Dynamic websites 6
2.3.1 Infinite scroll websites 6
2.4 Deployment 8

3 Analysis **11**
3.1 Coreference resolution 11
3.1.1 Saving Data 12
3.2 Sentiment analysis 12
3.2.1 Sentiment analysis with evaluation on neutral values . . . 13
3.2.2 Sentiment analysis neglecting neutral values 14
3.2.3 Sentiment analysis of summarized text 14
3.2.4 Considerations on sentiment analysis 14
3.3 Open information extraction 15

4 Conclusion **17**

1 Introduction

Mercurio is a project whose objective is to design and develop an integrated and modular system that draws information from various sources and uses it to predict the happening of out-of-the-ordinary financial events. To this aim the system integrates both time-dependent and highly frequent numerical data (e.g. price, volume) and textual data (e.g. financial news articles, financial breaking news). Data exposed by the sources are used to detect significant financial events that are either key-events (i.e. they convey considerable changes of the financial market) or signals (i.e. symptoms anticipating a key-event). Event recognition strategies vary depending on the type and nature of the managed data. The recognized events determine a temporal sequence of happenings that is then used to construct models to predict the happening of key-events in particular. To this aim sequential pattern mining techniques are applied to construct a model composed of temporal patterns. The constructed model and real-time data are used to provide users with alerts such as "there is a certain probability that company A will encounter key event C within X timeslots".
Until now, the Mercurio project has analyzed and considered data regarding the italian market and the companies that operates in it. The contribution that have been brought with our work regards, in the first place, the internalization of the sources of financial news by scraping the web, and secondly, some analysis have been implemented with a natural language processing software, that usually works better with english idioma, instead of italian. In particular, coreferences resolution, sentiment analysis and open extract information have been applied.

2 Web scraping

Web scraping is data scraping use for extracting data from websites. The term typically refers to automated processes implemented using a bot or web crawler. It is a form of copying, in which specific data is gathered and copied from the web, generally into a central local database or spreadsheet, for later retrieval or analysis [1].
The first target that had to be accomplished to turn the Mercurio project and its analysis to an international version was the data collection of financial news from various sources using web scraping techniques. In particular, the websites that have been considered were:

- bloomberg.com
- nytimes.com
- thisismoney.co.uk
- money.cnn.com
- marketwatch.com
- reuters.com
- moneymorning.com
- 4-traders.com

An important distinction among the sources mentioned above is whether or not is required an ajax/javascript interaction with the user in the web page that arranges the articles under consideration. Those that do not involve these interaction will be treated in the "static websites" section, the leftovers in the "dynamic websites" section, with a focus on infinite scroll websites.
To manage this issue of the project, the Scrapy open source framework [2] has been used. Once the project has been created, a single Spider [3] for each source has to be set up: they are the classes that define how a source will be scraped, and in particular, how to perform the crawling (i.e. follow links) and how to extract structured data from the page. The Spider's lifecycle varies a little between dynamic and static websites, but basically it starts by sending a request to URLs specified in starting_url, afterwards it gets the data, stores it on some physical support (e.g. database, files) and finally crawls to another source or invokes some javascript commands and repeats. The single steps could be more or less complicated depending on the considered source.
In order to make the storing stage work, the Scrapy Item Pipeline [4] has been used. It does a very simple task, once it receives an item that changes according to the source, it performs some action over it (i.e. store it).
Depending on the type of information that the website provides, the items that can be stored using the pipeline are two: "BriefItem", that contains a field for title, date, time, and an eventual URL where the news is better specified, and "NewsItem", which carries a value for title, authors, date, time, content and keywords. The former has been used when the use of the latter didn't make sense, due to a lack of information on the website, or due to the impossibility of retrieving it: for example, some websites (e.g. Marketwatch) provide a list of links to financial articles from many different websites, making out of the question to get the more specified data from such different website formats.

2.1 Saving the data

When it comes to saving the data there is a multitude of possibilities, ranging from a simple text file to a more structured database. At first, for testing purpose, the data has been saved in a tsv(i.e. tab separated values) textual file. Subsequently, in the deployment stage, it has been memorized on a database. For the management of the code it has been chosen to keep both the database and the files using a strategy pattern. This means that in order to test new scraping scripts in the future, it is possible to switch to the file methodology changing only few lines of code.

As it is shown on the architecture(quale architettura??), the item(i.e. BriefItem or NewsItem) goes through a pipeline, but before that, it must be loaded with data. The initialization of an item can be done with two methods:

- Creating an item with its constructor;
- Using an item loader to fill the field of the item.
  This kind of object lets the developer define two preprocessing operations for each field of the item:
  - input preprocessing, which provides a set of operations to be done when the data is filled inside the item loader (e.g. remove tags, remove escape characters, etc...)
  - output preprocessing, which provides an operation to be done when the item is created (e.g. taking only the first data of that specific field if there are a lot of data or concatenating those strings gotten through scraping)

While structuring and developing the scripts, no design pattern has been identified and used, except for the structure that the Scrapy framework itself already has. In fact the single steps that have to be performed have strong dependencies with the websites' structure and the ways that they arrange the information. For instance, most of the scripts are really light in term of lines of code, and the majority of them regards mainly the way in which the websites arranges its information and article (e.g. a set of <li> tags). The only model that came to mind is the Template method pattern [5], but it only introduces redundancy with the Scrapy framework structure.

2.2 Static websites

Nytimes.com and marketwatch.com are the only sources considered that doesn't fall under this category: every other websites doesn't require any kind of interaction with the client. In particular, the remaining ones can be further split into two categories: those that need to be scraped by modifying a value in the URL (i.e. the page number of a certain section) and those for which their sitemap can be used and analyzed to retrieve the links of the articles.

4-traders.com is a website which falls under the first of the two classes illustrated above. The parse method starts from a certain page number (zero) and retrieves the information required (e.g. links to various news); after that, another function that scrapes the single article is called. Finally, a request to the next page is done and the parse method will handle it.

The easiest way to scrape data from a website is via sitemap, but not every website has one. The sitemap is a special XML file containing a list of every "core" information of the website such as news, videos, stocks and so on; it is important for the website itself to have a well specified sitemap because huge crawlers like googlebot, yahoobot and other company's bots could use it to search for data and this could lead to a potential increase of its rank in the search results (Google also ranks web pages and not just websites). Of course, there are other benefits like:

- facilitating the work of the crawler;
- simplifying the categorization of content;
- reducing the amount of jobs that has to be done to monitor the visitors.

Another essential thing to take into consideration is the file "robots.txt", (every website has one) which contains all the rules a crawler should follow and sometimes there is also a sitemap. This is why before scraping everyone should examine this particular file.

This project, with the help of Scrapy, has searched through a lot of websites (Bloomberg, CNN, This Money,...) via sitemaps using an XML parser offered by Scrapy. Unlike dynamic websites, these are very simple to implement and that's why there are no particular ploy to use to increase the efficiency of the crawler. However, there is a big problem when someone scrapes data with nonchalance. The web company sometimes doesn't want every user agent to crawl its website, so they take countermeasures like temporary ban. For example, Bloomberg has implemented a detection system to identify unauthorized crawlers with the purpose of banning them; to solve partially this issue, it's possible to implement a rotation of the IP address synchronized with a rotation of the user agents. This method can partially prevent the detection, but for Bloomberg it just increases the number of GET request before getting banned. Nevertheless, the sitemap approach is still one of the best technique to scrape data.

2.3 Dynamic websites

Marketwatch.com and nytimes.com are the two sources requiring javascript interactions with the client. In order to handle the interplays, Selenium Python [6], combined with Scrapy, has been used: Scrapy alone may not be the optimal solution for these kind of issues [7].
Basically, Selenium allows the programmer to instantiate a WebDriver object, which is essentially a browser (it is possible to specify a preference among Firefox, Chrome, Android, PhantomJS, Safari and Opera) from which it is possible to execute javascript code. This is done, inside the Scrapy framework method "parse", once the page is specified and it's being scraped.

2.3.1 Infinite scroll websites

Scrolling was the type of interaction required with client in both websites mentioned above: an "infinite" page kept loading contents progressively while being scrolled. This led to various issues that have been faced and solved in different ways.

The only solution found on the internet with a brief research on different websites wasn't very smart w.r.t. used physical resources [8]: these simple scripts just run an infinite loop in which the driver executes some lines of javascript to scroll the page down. Although this could fit good for some application/websites that doesn't require nor make possible (due to the presence of shrunk content on the page) to execute many times the scroll, if many years of financial articles needs to be loaded it will lead to a waste of memory and a sensible slow down of the application, e.g.: after a year of news scraped for `https://www.nytimes.com/section/business/dealbook`, the Spider decelerated to the point where the javascript code was executed once every 5 minutes, while it took nearly 1 second in the beginning. This happens because the whole page is kept in memory (with many images displayed) and the DOM gets bigger at every request. Always referring to the previous example, after 1 year of news scraped, Firefox was using around 1.5 GB of RAM.
The first way we tried to bypass the problem, was using the browser in headless mode, that is without graphic interface, (to do this, an option of Selenium driver needs to be set up) but the issue persisted.
The second, and working, solution was to dynamically remove HTML code from the page while it's being scraped (i.e. by executing javascript removing scripts with Selenium). This has been done in different ways that are going to be compared later in this report.

The first website which implements infinite scrolling is `https://www.nytimes.com/section/business/dealbook`. The solution adopted for scraping the nytimes is based on opening the website in device-mode (i.e. with browser in mobile-mode): in this way, instead of scrolling, a "Show more" button is displayed. In fact, if the website is opened in desktop mode the button "Show more" disappears after the first time it is clicked and the infinite scrolling knocks in, while in mobile mode it is persistent. Once the scraper sets up the correct way of opening the website, it removes the unnecessary static HTML code from the page and a while true loop starts. The algorithm, at first, clicks on "Show more", then it takes the new articles loaded and it provides them to the pipeline for saving them. Subsequently, the script for removing all the <li> tags is executed in order to reduce the physical resources used by the page. Finally, the button is clicked again and the cycle is repeated. However, when the script scrapes more than two years of financial news, it becomes a bit slow.

The solution adopted for `https://www.marketwatch.com/newsviewer` works as follow: the content scraped with this Spider is the one displayed in the sidebar at the top of the page indexed by the link given. As for the case of nytimes.com, at first the unnecessary static HTML code of the page is removed by executing some javascript lines. After that, a while true loop is entered, in which the driver executes the code necessary for the scroll: every 10 iterations, the interesting data is retrieved and the various Item objects that weren't analyzed before are provided to the Pipeline; furthermore, a script that removes each <li> tag is executed: in this way, the usage of physical resources is constant w.r.t the volume of information scraped. This happens because the rise of quantity of <li> tags and the dimension of the DOM are programmatically limited by the algorithm, which removes data that has already been scraped.
The number 10 has been chosen empirically: indeed, in this solution there were many problems related with the deletion of HTML contents and how the ajax functions calls restore them and use them to generate the older articles. For

instance, it's been tried to delete all the tags, or a part of them such as an half, in each iteration, but that wasn't working because they were re-established instantly by the ajax functions, thus slowing down the whole process: more specifically, the majority of the contents were restored, thus leading to analyze the same contents many times. Doing that each 10 iterations allowed to keep a good velocity because not the totality of the articles were spawn; however, the revival of deleted news still needs to be handled: the timestamp of the latest article stored on physical support is saved: the only news that will be written are the ones previous to that timestamp. This is necessary because the ajax calls restore <li> tags that analyzed in the previous iteration.

Despite having already found a feasible solution for the marketwatch.com website, we identified another another solution: the content scraped in this case is in the sidebar at the bottom of the page. The technique used this time is called "javascript injection" and the process is composed of two parts:

- First of all, it's necessary to analyze the code written by the website developer to find the code that sends a request to the server to get the oldest news. The javascript code is accessible by the browser using "View Source".
- After finding the code and injecting the javascript executable command, it is possible to get the old news.

This method avoids many of the problems about the infinite scrolling and can also exploit the normal update of the news (e.g. asking the server to get a chosen number of news instead of a the fixed one determined by the website developer constraint) but it's not ideal. In this solution, while analyzing the javascript code, it has been detected a fix for the reloading of the news; Basically this website every 60 seconds load all the newer news only if the height of the sidebar page is at the top, but for the objective of scraping old news, these news are trivial. So the trick or solution adopted , when deleting the news already scraped, is to leave a good amount of news and scroll the sidebar page at the half of its maximum height, so the javascript written by MarketWatch doesn't work.

The study of the code written by someone else is quite a difficult task to do, because it depends on the skill of the counterpart programmer and it's not always possible to find the code that calls the server. All in all, despite this technique being quite trivial, revealed to be efficient w.r.t. the two previous methods.

2.4 Deployment

After finishing the development of the spiders, it's necessary to deploy the project on a server. For this, the protocol SSH(Secure Shell)[11] has been used to access the server and create the table of the data in the dabase. Since the server is accessed by more than one person, it has been decided to use the command 'screen' to create a new thread screen that keeps running on background even if the connection to the server is closed. For the database it has been used MySQL and a python library called MySQLdb; the queries used are:

```
CREATE TABLE articles_en_full(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    date DATE,
    time TIME,
    title VARCHAR(100),
    newspaper VARCHAR(30),
    author VARCHAR(50),
        content TEXT,
    tags TEXT,
)

CREATE TABLE articles_en_partial(
    id INT NOT NULL PRIMARY KEY AUTO_INCREMENT,
    date DATE,
    time TIME,
    title VARCHAR(255),
    newspaper VARCHAR(100),
    url VARCHAR(200),
)
```

When deploying crawlers, it's very important to make use of a server instead of client machines: this is due to the fact that servers have more stability in terms of connection and computing power. This is why the scraping project needs to be deployed on a server; to solve this issue "scrapy" offers an application for deploying and running Scrapy spiders, called scrapyd[9]. This daemon, background process, is executed by using screen and the command 'scrapyd' on a server waiting for messages on a specific port. Another feature offered by scrapyd is a set of JSON API to deploy the projects and scheduling the spiders; for example:

- daemonstatus.json, which returns the status of the scrapyd application that is running on the server
- addversion.json to deploy a project
- schedule.json to schedule a spider
- cancel.json to stop a spider

One more way to deploy the project is to use an application client for scrapyd, called scrapyd-client[10]; but this doesn't provide all the API operations that scrapyd offers natively. The solution adopted for Mercurio is a mix of both techniques. For the deployment of the project it has been used the command of scrapyd-client while for the canceling of the spider it has been used the JSON API cancel.json.

Before deploying the project on the server, it's important to define what package, folders and files are necessary and then keep going with the deployment; this is defined by the file setup.py auto generated by scrapyd. In this project, the scrapyd daemon is running on a server of the Politecnico of Milano listening on port 6800. In many cases, the server could be unreachable from the outside

(e.g. internet) due to the presence of firewalls, so before deploying, it's necessary to make a port forwarding to the server via SSH protocol. After deploying the project it's possible to schedule spiders in order to crawl data and store it; to check the scheduled spiders and their logs, scrapyd offers a minimal GUI web interface accessible through browser at localhost:9000.
A server owned by Politecnico di Milano has been used to host scrapyd, used with the support of scrapyd-client, and to save the data on a MySQL database.

3 Analysis

As said before, after the data collection process, the information has been analyzed and manipulated in different ways that are going to be explored in the following pages. In order to accomplish this task, Stanford CoreNLP [12] has been used. In particular, a python package that uses Stanford CoreNLP server, has been adopted [13].
This natural language processing software provides a set of human language technology tools that makes very easy to apply linguistic analysis to a piece of text (in this case the financial article).
The python wrapper presents a simple API that allows the programmer to call the Stanford CoreNLP annotators [14], which basically process a sentence in a certain language returning a response in a specified output format, e.g.: JSON, XML or text. The following example, performs the sentiment analysis of a given string and stores it into an XML variable.

```
text = "This is an example"
props = {'annotators': 'sentiment','pipelineLanguage':'en','outputFormat':'xml'}
xml = nlp.annotate(text, properties=props)
nlp.close()
```

Among the available types of output, XML has been chosen: we parse it with Beautiful Soup 4 [15].

Coreference resolution, lemmatization, sentiment analysis and open information extraction, are the main language tools that have been used. As a result of the fact that dcoref, which is the annotator that solves pronominal and nominal coreferences, and lemmatisation takes significant amount of time to run, their manipulated results are saved in two different fields of a new database table. Doing so allows to change the mechanisms of the scripts regarding sentiment analysis and open information extraction in future, and therefore re-run them without the time waste mentioned above.

3.1 Coreference resolution

The coreference occurs when two or more expression inside a text refers to the same object (e.g. "Jessica has sold her phone to Marcus": in this phrase the pronoun "her" refers to "Jessica"). This analysis is done to capture every coreferences inside the article scraped from the website, with the help of the external library Stanford CoreNLP, and substitute it with the representation of that expression (e.g. Jessica has sold Jessica's phone to Marcus).

Basically, the algorithm written for this analysis consists in dividing the text in more phrases splitted by a dot, and for each phrase analyze it with CoreNLP. After that it has been used BeautifulSoup to analyze the XML which the external library gives back; to find the representation and the coreferences, there is a tag "coreference" inside the XML. Since there is also another tag called "sentences" that contains all the text analyzed, so it has been used for the substitution of the pronoun or other literal expression into the representation text. Every time a coreference expression is founded, the algorithm will replace that coreference expression (i.e. pronoun or other literal expression) with the

representation of that. At the end the entire text is reconstructed from the tag XML "sentences" that contains the string of text replaced.

It was also possible to analyze the entire text with CoreNLP, but after some tests it has been founded a lot of errors about the coreference (e.g. replacement of "he" with "Clinton"). So it has been decided to split the entire article into different substrings of 5 phrases and with this technique it's possible to at least found coreference inside two or more expression.

3.1.1 Saving Data

After the analysis of coreference the new data needs to be saved somewhere. In this project it has been stored on a database. The tables created for all the study of article content are:

- articles_en_analyzed: this contains the same field of the old table "articles_en_full" but with 5 more fields:
  1. coref_content: a field that contains the new article content with the coreference replaced with its representative
  2. lemma_content: this contains the text article with every word lemmatized (made by using the lemma annotators)
  3. sentiment: a field that can contains "very negative", "negative", "neutral", "positive" or "very positive". This is useful for the sentiment analysis
  4. sentimentNoNeutral: it's the same of the above one, but in this case the "neutral" is ignored (i.e. the "neutral" sentiment returned by a phrase from CoreNLP is ignored)
  5. sentimentSummarized: it's the same of the first sentiment, but with this, it analyzes a summarized text instead of the entire article
- openie_reports: this table is linked with the article id of the table above, and it contains 5 fields:
  1. reportId: the number identification of the openie_reports (primary key)
  2. articleId: the article id of the table above (it's a foreign key)
  3. subject: it represents the subject inside a literal expression
  4. verb: it represents the action of the subject
  5. object: it represents the object on which the action is taken

In this part of the project, all the news articles are read from the table "articles_en_full", and then after the analysis, the news is saved on the new table "articles_en_analyzed" with two new field: coref_content and lemma_content.

3.2 Sentiment analysis

Sentiment analysis refers to the use of natural language processing and text analysis to detect automatically, extract, quantify and study states and subjective information. It aims to determine the attitude of a writer/speaker w.r.t

some topic or the overall contextual polarity or emotional reaction to a document, interaction, or event [16].
The purpose of our analysis is to give each article a sentiment, in order to allow pattern mining to forecast financial catastrophes. This approach has been implemented in the english version of the project because of the well-trained nlp libraries available and due to the fact the international press was thought to be more vigorous and less apathetic compared to the italian one.
The sentiment tool provided by Stanford CoreNLP parses a sentence and returns "very negative", "negative", "neutral", "positive" or "very positive", depending on the deep learning model [17] used in its implementation. So it's been necessary to find a way of weighting the results associated with each sentence found in the article.
The problem has been approached in three different ways that will be explained in the following pages. However, every implemented solution is based on the fact that clearly not all of the phrases in an article have the same level of importance, thus some periods should weight more than others. The solution adopted to overcome this problem is creating a lemmatized vocabulary composed of financial nouns and keywords. When parsing a sentence, the more it contains these terms, the more it weights, following a linear relation.
$weight = 1 + 0.5 * n$
Where n is the number of words present in the dictionary that appear in the sentence too. The multiplier coefficient (0.5) has been chosen in an empirical way and can be subject to further research and learning models.
The weight is then used to modify a python dictionary which has as keys "very negative", "negative", "neutral", "positive" and "very positive" according to the result given by Stanford CoreNLP sentiment annotator.

3.2.1 Sentiment analysis with evaluation on neutral values

Due to the fact that studies have reported the importance and relevance of neutral values [18], in this first method they are considered not negligible. Furthermore, the whole text of the financial news is considered.
So, once that the algorithm described above is applied, when it comes to assigning the sentiment of the article, the dictionary comes into play: the aim is to determine a number between 1 and -1 that represents the sentiment of the article (w_a). The interval [-1,+1] is then divided in five equals parts, each of which is assigned to a sentiment and according to w_a the article's sentiment is calculated.
Let be w_{vn}, w_n, w_{ne}, w_p, w_{vp} the weights of "very negative", "negative", "neutral, "positive" and "very positive" for the whole article respectively, and m_{vn}, m_n, m_{ne}, m_p and m_{vp} their central position in the [-1,+1] interval. Being i the position of the sentiment mentioned above:

$i \in N, i \in [0, 4]$

$$m_i = -0.8 + i * 0.4 \tag{1}$$

$$w_a = \frac{\sum_i w_i * m_i}{\sum_i w_i} \tag{2}$$

Doing so, some important properties are guaranteed: neutral values are relevant and w_a gets closer to the most valued voice of the dictionary. An important note is that this approach won't permit to compare different level of positiveness and negativeness.
Nevertheless, once that the script's been run on the server, the results didn't match the expectations: the majority, or, for better saying, almost the totality, of the articles were valued as "neutral" due to the fact that a huge number of phrases contained were analyzed "neutral" as well from NLP. This is the main reason that lead to the development of the second and the third methods.

3.2.2 Sentiment analysis neglecting neutral values

The second method that has been studied neglects the presence of neutral values. The only thing that is going to change w.r.t. the precedent solution, is

$$w_a = \frac{\sum_i w_i * m_i}{\sum_i w_i} \tag{3}$$

In this case in the denominator the summation doesn't involve w_{ne}. In this way, the huge amount of weight of neutral sentences is nullified, thus making the algorithm much more susceptible to the presence of positive or neutral phrases.

3.2.3 Sentiment analysis of summarized text

The last solution makes use of a summarization script, which was developed for the italian version of Mercurio and that has been adjusted for the english version by modifying the list of stop words. The mentioned summarization script makes use of a combination of extractive and abstractive summarization methods. The former consists in selecting important phrases and concatenating them into a shorter form by means of statistical and linguistic features, while the latter adopts linguistic methods to examine and interpret the text in order to understand the main concepts in a document. The result contains the 25% of the original sentences.
Due to the fact that a summary contains the most important sentences of an article, analyzing the abstract of a financial news by removing the most of the content could benefit the result because, as already said, not all of the phrases present in it weights the same.
The summarization has been combined with the strategy described in the first method.

3.2.4 Considerations on sentiment analysis

For instance, consider the following sentence: "Glaxosmithkline has offloaded its rare diseases unit as chief executive Emma Walmsley continues to trim the business. Britain's biggest drug maker sold the portfolio to Orchard Therapeutics, a private biotech firm with bases in London, Boston and California. It comes after Walmsley said the unit was under review last July, as part of a wider effort to refocus the company's research on the areas of respiratory, HIV and infectious diseases, cancer drugs and immuno-inflammation. The deal is

small financially for Glaxo but will see it take a stake of nearly 20 per cent in Orchard.". With all the possible methods, the result is "neutral". However, by querying the database it is possible to infer some conclusions. The efforts for making the algorithm more susceptible to positive and negative phrases were totally useless: all the row of the three columns saved in the database reports only "neutral" result. This is due to the fact that the initial hypothesis that american press isn't apathetic is basically false. In fact, the script has been launched printing the sentiment of each analyzed sentence, and they are all reported as "neutral". This is clearly a problem for the entire process.
Furthermore, independently on the method considered, there is a problem with the whole algorithm that regards linguistic itself. Let's consider the phrase "Facebook is near to declare bankrupt due to the recent scandal. Its competitors can take a big advantage from this.". The first period is valued "negative", while the latter "positive", so, a further implementation has to consider this fact: the sentiment depends even on the subject of the period: some fact could carry a benefit to a company at the expanse of its competitor, for instance. A future work could be developing the sentiment analysis for only a small group of companies, and implementing an entity recognizer for the articles and the single sentences, in order to calculate in a more precise way w_a. The focus on a small group of companies allows, furthermore, to associate each firm with its CEO, in a deterministic way.

3.3 Open information extraction

The Open Information Extraction (OpenIE) annotator extracts open-domain relation triples, representing a subject, a relation, and the object of the relation. This allows to summarize the articles in a series of triples and to evaluate the actions performed by the subjects. The algorithm, in order to work, requires that the articles are first pre-processed, by resolving the co-references so as to extract the triples with the correct subjects. The process begins by asking in input the reference journal from which to retrieve the financial items to be examined, then subdividing the individual articles into short periods that are easier to analyze from the Stanford coreNLP library. The library for each period returns a possible triplet that summarizes the described action, and assigns to the analysis a degree of confidence that goes from 0 to 1, where 1 is the maximum correctness. This is one of the critical points of the algorithm, since there are several solutions that have a degree of confidence at 1, choosing the most correct is a very complex problem that would require a much more specific analysis, which at the moment is impossible to find in any open source library. For this reason simply returns the first triplet with confidence to 1 that it finds.

```
text = "This is an example"
props = {'annotators': 'sentiment','pipelineLanguage':'en','outputFormat':'xml'}
for triple in list_of_solutions.find_all("triple"):
            if triple.get("confidence") == "1.000":
                for text in triple.find_all("text"):
                    triple_save = triple_save + text.string + " "
                if len(triple_save.split()) == 3:
                    return triple_save
```

The idea behind this analysis is to evaluate the temperature of the verbs, that refer to the financial sector such as buy, sell, invest etc... found, that is counting the number of times in which the same actions are presented in the various articles and based on their number, to risk a hypothesis on the progress of the subject who performs those actions. If a company buys a lot in the last period, the company is probably doing well. Although the idea was good, this analysis results to have many different problems:

- The co-references very often are not correct and the pronouns are not replaced with the correct subjects.
- The openIE analysis of the periods often fails to extract a triplet or the result don't have the confidently to 1.
- Many verbs extracted from the triples are not contained in the financial dictionary.

For all this the analysis on the whole is not very correct. The library turns out to be much more effective on simple sentences or on articles related to politics, where it manages to study the structure of the sentence and to return satisfactory results. We hope that in the future will come out more powerful open source tools for this type of analysis on natural language.

4 Conclusion

Thanks to this work, now the Mercurio project has new data collections available, concerning the foreign press in the financial sector, on which in the future it will be able to perform more specific analyzes, thus opening the doors to an analysis of the stock exchange on a wider scale. Furthermore, the scripts for scraping lay the foundation for many other scripts of this type, facilitating the writing and speeding up the phase of acquiring information. The analysis work, instead, shows the critical issues of the open source coreNLP library in the analysis of the natural language that will certainly require much more knowledge in order to obtain satisfactory results.

References

[1] What web scraping is, URL https://en.wikipedia.org/wiki/Web_scraping

[2] Scrapy framewrok, URL: https://scrapy.org/

[3] Scrapy spider, URL: https://doc.scrapy.org/en/latest/topics/spiders.html

[4] Scrapy pipeline, URL: https://doc.scrapy.org/en/latest/topics/item-pipeline.html

[5] Template method pattern, URL: https://it.wikipedia.org/wiki/Template_method

[6] Selenium Python, URL: http://selenium-python.readthedocs.io/

[7] Web scraping framework review: Scrapy vs Selenium, URL: https://blog.michaelyin.info/2017/11/06/web-scraping-framework-review-scrapy-vs-selenium/

[8] Web scraping for infinite scroll: URL: https://michaeljsanders.com/2017/05/12/scrapin-and-scrollin.html

[9] Scrapyd http://scrapyd.readthedocs.io/en/stable/#

[10] Scrapyd Client https://github.com/scrapy/scrapyd-client

[11] SSH Secure shell, URL: https://en.wikipedia.org/wiki/Secure_Shell

[12] Stanford CoreNLP, URL: https://stanfordnlp.github.io/CoreNLP/

[13] Python wrapper for Stanford CoreNLP, URL https://stanfordnlp.github.io/CoreNLP/annotators.html

[14] Stanford CoreNLP annotators, URL: https://stanfordnlp.github.io/CoreNLP/annotators.html

[15] Beautiful Soup 4, URL: https://www.crummy.com/software/BeautifulSoup/bs4/doc/

[16] What sentiment analysis is, URL https://en.wikipedia.org/wiki/Sentiment_analysis

[17] Deep learning for sentiment analysis, URL: https://nlp.stanford.edu/sentiment/

[18] The importance of neutral values in sentiment analysis, URL: http://blog.datumbox.com/the-importance-of-neutral-class-in-sentiment-analysis/