IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

KNOWLEDGE EXTRACTION FOR AUTOMATED RECRUITMENT PROCESS USING NATURAL LANGUAGE PROCESSING

[1]Sushit Kumar, [2]Shalini Bhadola, [3]Kirti Bhatia

[1]M.Tech Student, [2]Assistant Professor, [3]Assistant Professsor
[1]Computer Science & Engineering,
[1]Sat Kabir Institute Of Tehnology & Management, Bahadurgarh, Haryana, India

Abstract: When companies post jobs to their sites or other sites then hundreds of applications are received. Reading these CVs and recruiting the right candidates is a work that will include higher workforce to getting the result manually. Parsing Resume and scanning the same is an old process and estimating his proficiency in certain areas are practiced earlier. If we see currently there are no open-source CV parsers are available to extract knowledge from CV with expected results and therefore the purpose of the dissertation is to remove the shortcoming in the current Recruitment System's which are used for extracting information from available CV that was available with the organization. With this research dissertation we are trying to generate solution to problem and provides accuracy in results and does the work faster. CV/Resumes taken from different professions use different formats, fonts, styling and structuring and vocabulary. We are focusing the concept of Automated Recruitment process and the mail logic behind it is to provide ease to Recruitment Process with the help of Technology.

1. INTRODUCTION

Natural Language Processing (NLP), also known as Computational Linguistics (CL) can be defined as the automatic processing of human (written or spoken) language (Copestake, 2004). The translation of human speech research and text reaches to the early 1950s. The first public demonstration of Machine Translation (MT) was the IBM-Georgetown Demonstration of 1954, where Russian text was translated to English (Jones, 1994).

Since then many other directions of NLP emerged, such as grammar and spelling correction, text categorization, summarization, question answering, speech recognition and information extraction. However throughout years the technology and its diversity, and the application areas and the quality of the analyzed text changed. In Twitter posts, where texts are short, noisy and context dependent, the information extraction is even harder (Derczynski, Maynard, Rizzo, et al., 2015). CVs contain incomplete listings and incomplete sentences so extracting information in this domain is not easy.

2. STEPS OF KNOWLEDGE EXTRACTION

2.1 Sentence segmentation

2.1.1 Corpus

In NLP and linguistics it is common to refer to a large amount of structured texts as "corpus" (plural corpora) (Bird, Klein, & Loper, 2009). Different corpora contain large amount of texts from different domains. For example the American National Corpus1 contains over 15 million words from transcripts and spoken data produced since 1990. Additionally corpora (such as the Brown Corpus2) often contain additional linguistic information ("grammatical annotations") to the text that describe the linguistic role of each word in the sentence (e.g.: noun, verb etc.). When developing statistical methods for NLP, these are often trained on such corpora.

2.1.2 Rule-Based Approaches

A straightforward approach to solve this problem would be to introduce a short list of sentence-final punctuation marks such as "?", "." and "!", however such an approach starts to malfunction when common mistakes such as "etc." or "e.g." appear in the text (Reynar & Ratnaparkhi, 1997). Then one could create an exception list that would contain words where the dot as a punctuation sign is disregarded, however such rule-based lists will be never complete , moreover multiple rules might interact erroneously with each other. For example, what happens when the sentence ends with an abbreviation such as "Mr." or "Mrs."? In such cases the punctuation marks the end of the abbreviation as well as the end of the sentence.

2.1.3 Supervised Machine-Learning Approaches

Riley (Riley, 1989) introduced a decision tree to classify the punctuation marking the sentence boundaries using the following features:

- Probability of the word preceding e.g. "." occurs at end of sentence
- Probability of the word following e.g. "." occurs at beginning of sentence
- Length of word preceding e.g "."
- Length of word after e.g "."
- Case of word preceding e.g ".": upper, lower, or number
- word after ".": upper and lower or number
- Punctuation after "." If any

Riley's method captures information from both sides of the punctuation and calculates that a certain word occurs before or after a punctuation mark. This method was trained by him his method on 25 million words of AP newswire[1] corpus.

2.1.4 Unsupervised Machine-Learning Approaches

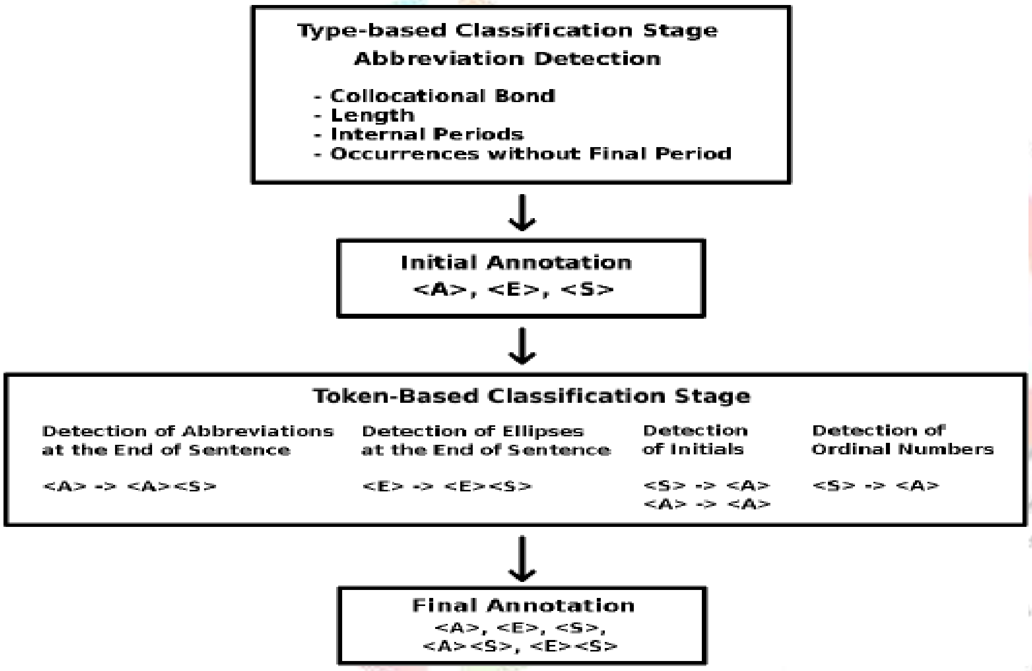


Figure 1: Architecture of the Punkt System

In contrast to supervised machine-learning approaches, unsupervised approaches don't require a hand-annotated test data to be trained on. the extra information that's required for the sentence boundary detection is extracted from the test data itself. Kiss proposed an answer for sentence boundary detection in 2006 (Kiss & Strunk, 2006)

Abbreviations often contain word-internal periods. (For example in "U.S.A" or in "e.g.".) The process starts with the classification of type-based. as an example a punctuation and also the preceding form a good collocation if the probability of the punctuation occurring, given the preceding word (P(punctuation w)) is a minimum of 99%.

2.2 Tokenizing

Rule based tokenizers use a hard-coded list of rules, when classifying characters or blocks of characters as tokens. The widely used Penn Tree Bank (PTB) tolenizer1, is an example of rule based tokenizers. The PTB tokenizer uses an inventory of pre-coded Regexps (López & Romero, 2014) that tell the algorithm a way to proceed once a matched expression is found. To demonstrate its principle, Listing 2.2 shows a simplified code-snippet from the Python implementation of the PTB Tokenizer2.
In among the statistical approaches Hidden Markow Model are also known for development of NLP applications". A HMM is a automation of finite-state stochastic state transitions and symbol emissions" (Rabiner, 1989). Jurish and Würzner presented a tokenizer in 2013 which makes use of HMMs to determine the boundries of word and sentences. In their approach the tokenization was divided in two steps:

- Scanning: A so-called "scanner" was applied that split the raw input text on whitespace and punctuation.
- HMM Boundary Detector: The HMM applied in their paper had 6 observable features such as length of the segment, latter case

of the segment etc. and it had 3 hidden binary features (beginning-of-word, beginning-of-sentence, end- of-sentence). The HMM took the output of the scanner and applied the Viterbi algorithm (Viterbi, 1967), which returned the optimal state sequence, thus the most optimal values for the hidden features of the HMM.

2.3 Part Of Speech Tagging

Many words have different meanings and in such cases it's beneficial to possess additional syntactical information on the text. as an example the sentence
They can fish.
can be interpreted in two ways:
• touching on a gaggle of people, who have the flexibleness to fish,
• concerning a gaggle of people, who are currently putting fish into a can.
The actual meaning of the word is indicated with additional POS tags. For this purpose the common tagging-schemes are the Penn Treebank Tagset (PTT) and also the CLAWS 5 (C5) Tagset2. with using the C5 schemes, the words of the above mentioned

sentence is also tagged as follows:
they PNP
could VM0 VVB VVI NN1 fish NN1 NN2 VVB VVI

PPN (personal noun)
VM0 (modal auxiliary verb) VVB (base type of verb)
VVI (infinite reasonably verb) NN1 (singular noun)
NN2 (plural noun)
The POS tagger's task is to predict the foremost possible (and plausible) interpretation of the sentence, that is:
They (PNP) can (VM0) fish (VVB) .(PUN)

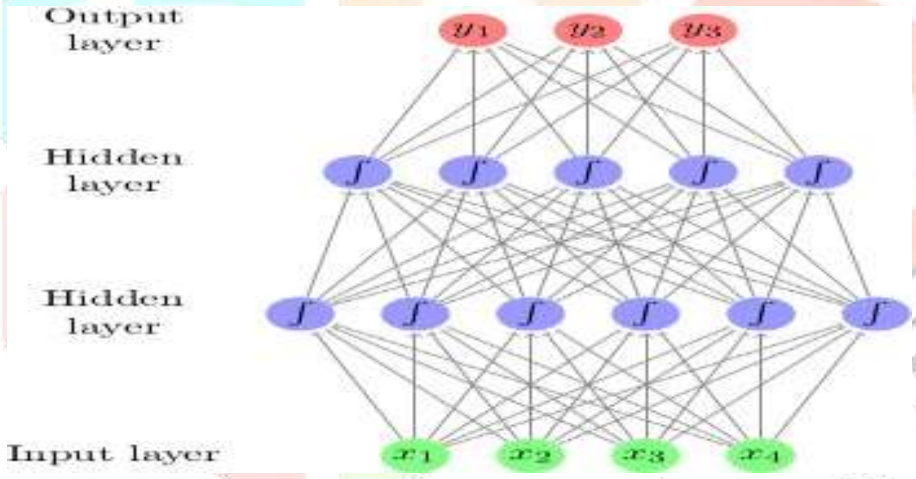


Figure 2: Feed-forward neural network with two hidden layers. (Goldberg, 2016)

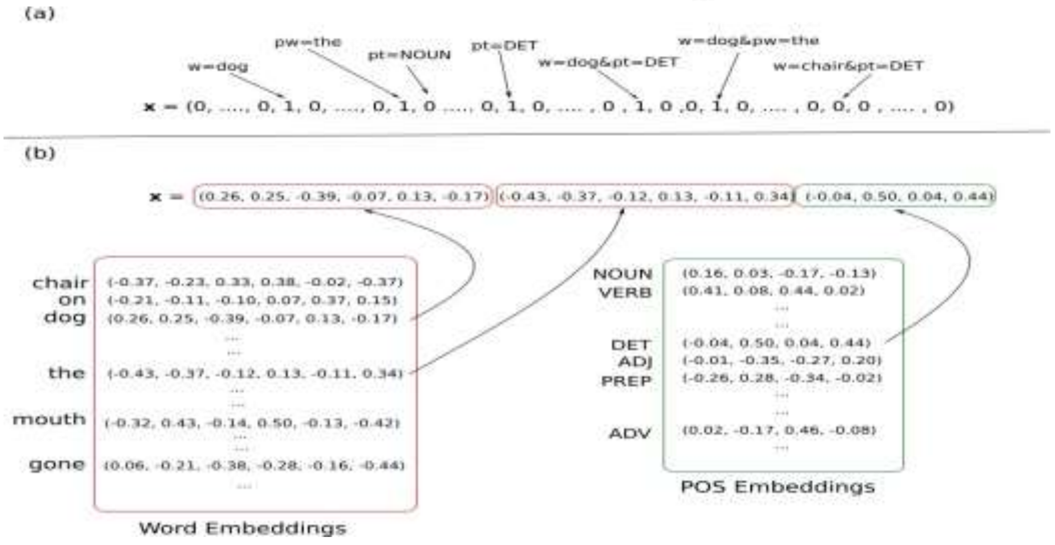

(a)
pw=the
pt=NOUN
pt=DET
w=dog&pw=the
w=dog
w=dog&pt=DET
w=chair&pt=DET
x = (0, ..., 0, 1, 0, ..., 0, 1, 0 ...., 0, 1, 0, ... , 0 , 1, 0 ,0, 1, 0, ... , 0, 0, 0 , ... , 0)
(b)
x = (0.26, 0.25, -0.39, -0.07, 0.13, -0.17) (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34) (-0.04, 0.50, 0.04, 0.44)
chair (-0.37, -0.23, 0.33, 0.38, -0.02, -0.37)
on (-0.21, -0.11, -0.10, 0.07, 0.37, 0.15)
dog (0.26, 0.25, -0.39, -0.07, 0.13, -0.17)
the (-0.43, -0.37, -0.12, 0.13, -0.11, 0.34)
mouth (-0.32, 0.43, -0.14, 0.50, -0.13, -0.42)
gone (0.06, -0.21, -0.38, -0.28, -0.16, -0.44)
Word Embeddings
NOUN (0.16, 0.03, -0.17, -0.13)
VERB (0.41, 0.08, 0.44, 0.02)
DET (-0.04, 0.50, 0.04, 0.44)
ADJ (-0.01, -0.35, -0.27, 0.20)
PREP (-0.26, 0.28, -0.34, -0.02)
ADV (0.02, -0.17, 0.46, -0.08)
POS Embeddings

Figure 3:Comparison of sparse (a) and dense feature (b) representation (Goldberg, 2016)

3. PROPOSED SOLUTION AND RELATED WORK

Approach

Each iteration consisted of the following steps:

- Defining a set of features for the iteration.
- Creating a product increment that satisfies the previously defined features.
- Evaluating the performance of the complete product, after the features were integrated into it.

Required Features

The goal of the first iteration was to create a minimum viable product, in this case, a software, that fulfills the following requirements:

- Given a folder containing resumes, the software should be able to parse each resume, and extract the candidate's work experience and skills.
- For each work experience the description and the corresponding start and end dates are extracted.

Implementation:

Text extraction without losing structural and visual information

The major problem with the currently existing open-source parsers is, that they only extract the raw text from the resume, losing the other structural and visual information on text-color, font-size, and text alignment. While pdf2htmlEX does an excellent job at converting PDFs to HTML, it's three serious drawbacks:

• It is an external program, that needs to be separately installed on the pc, thus the resulting solution will have an external dependency.

• The HTML files generated by pdf2htmlEX are dynamic, meaning that they contain JavaScript code that renders the particular Document Object Model of the page. When the user scrolls to the given page. This makes text and data extraction harder, because the more popular scrapers don't have integrated browser to render the content of the page. so as to extract information from dynamically rendered HTML files, a web-driver based scraper has got to be used, like Selenium2.

• Using web-driver based scrapers have a large impact on the general parsing time, because it takes plenty of computational resources to start out the web-driver engine scroll to every page individually, dynamically render their content, and extract the visual and structural information for every line.

Given the HTML file, the question whether which tool to use so as to extract structural and visual information continues to be opened. After an intensive research on the subject, the followings are often concluded:

• The selected tool for information extraction mainly depends on the programing language used.

• Articles that debate scraping with Python use Selenium.

• Articles that debate scraping with JavaScript (Gil, 2015; Barr, 2016) primarily use PhantomJS3 and Scraperjs4.

Since the present project is predicated on Python, Selenium was chosen for scraping the HTML file. The resulting text extraction process may be summarized within the following three steps:

• In order to avoid opening and rendering the HTML file, anytime a line's text or visual information is required, each line's text, visual and structural properties are extracted from the HTML and stored in an object. this is often similarly to the Flyweight design pattern described by Gamma (Gamma, 1995).

In the end of the method the program has an "internal representation" of the resume, in type of an inventory of objects. Each object contains the text of one line and its visual properties like font-type, left-margin, horizontal position. the method is visualized on Figure 11.

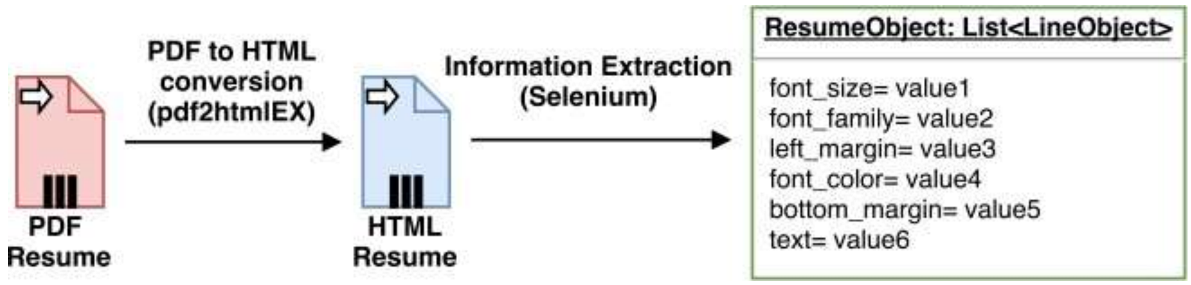


Figure 4: Visualization of the information extraction process

Analyzing the resume

Having the text of the resume and its visual content, the subsequent step is to interrupt up the resume into sections (such as work experience, education and skills). For instance the section keyword "skills" would have the synonyms "skills", "abilities", "computer knowledge" within the predefined dictionary. Finally the parser would seek for occurrences of those keywords within the resume, and also the resume is separate into sections, along the successfully found keywords. The main disadvantage of such approach is that it doesn't recognize new keywords, and rather than disregarding unknown sections it keeps appending its content to the text of the last successfully identified keyword's section. This ends up in a "polluted" output, where one section (e.g.: Education) contains text that originally didn't belong to the present section.

In order to resolve the matter of identifying unknown keywords, the resume is first sought for known section keywords. Then the successfully recognized section keyword's visual properties are analyzed. supported these visual properties, the algorithm makes a final assumption on the visual and structural properties of section keywords. That the yet unknown section keywords also are written with this styling. the method is visualized on Figure 12.

Breaking up resume into sections

In this stage the resume is broken up into sections, along the successfully found section keywords. The resume is interpreted line by line, however each line's visual properties are compared against the previously assumed visual properties of section

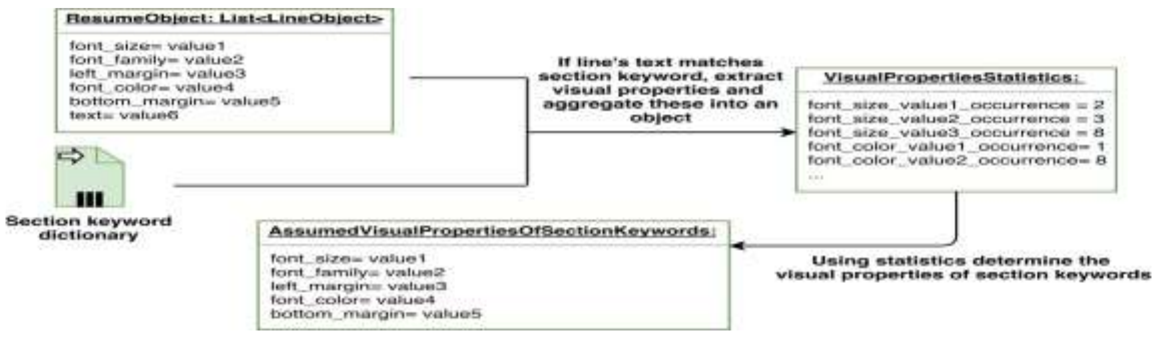

ResumeObject: List<LineObject>
font_size= value1
font_family= value2
left_margin= value3
font_color= value4
bottom_margin= value5
text= value6
If line's text matches section keyword, extract visual properties and aggregate these into an object
VisualPropertiesStatistics:
font_size_value1_occurrence = 2
font_size_value2_occurrence = 3
font_size_value3_occurrence = 8
font_color_value1_occurrence= 1
font_color_value2_occurrence= 8
...
Section keyword dictionary
AssumedVisualPropertiesOfSectionKeywords:
font_size= value1
font_family= value2
left_margin= value3
font_color= value4
bottom_margin= value5
Using statistics determine the visual properties of section keywords

Figure 5: Deduction of structural visual properties of section keyword

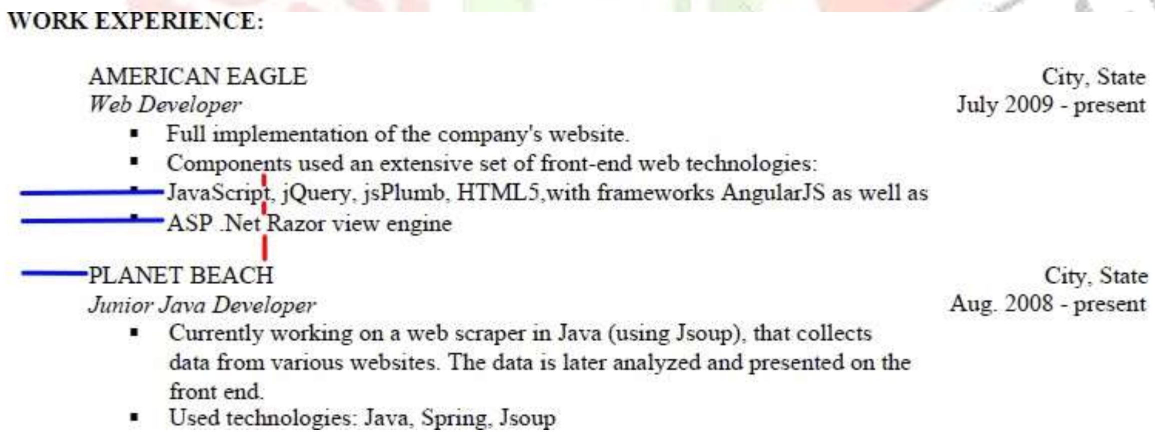
This way when an unknown expression comes, seeing its visual properties the algorithm can make a more educated guess, whether the expression is a section keyword (and hence its content belongs to a separate section), or if the text belongs to the previous section. At the end of this stage, the resume is broken into sections, along the section keywords.

Breaking up sections into experiences

Each section contains several individual experiences. In case of work experiences each individual experience corresponds to a company and a time interval, where the candidate has previously worked. Breaking up sections into individual experiences is a harder task than breaking it along section keywords, as separate experiences are rarely marked with different visual properties. Browsing through numerous resumes it can be concluded, that people often use horizontal spacing (e.g. enter) and vertical spacing (e.g. "tabbing" or "vertical space from the left border of the page") to visually differentiate between sections. On Figure 13 it can be seen, that the horizontalspacing (red) increases, as a new experience section begins, meanwhile the left-margin (blue) is reduced.

Using the horizontal spacing between lines and the left-margin of each line, the algorithm implemented in this iteration identifies a new section, if the following two criteria are met:

- The left margin of the current line is smaller than the left-margin of theprevious line.
- The horizontal spacing between the current and last line is greater than the horizontal spacing between the last line and the one before.


WORK EXPERIENCE:

AMERICAN EAGLE City, State
Web Developer July 2009 - present
- Full implementation of the company's website.
- Components used an extensive set of front-end web technologies:
- JavaScript, jQuery, jsPlumb, HTML5,with frameworks AngularJS as well as
- ASP .Net Razor view engine

PLANET BEACH City, State
Junior Java Developer Aug. 2008 - present
- Currently working on a web scraper in Java (using Jsoup), that collects data from various websites. The data is later analyzed and presented on the front end.
- Used technologies: Java, Spring, Jsoup

Figure 12: Distinguishing section using tabbing and horizontal spacing

Meanwhile this approach uses the structural and visual information of the resume it didn't work for one-third of the 25 resumes used for testing, mainly because these people don't use this kind of visual structuring.

Identifying dates

Once the individual work experiences are identified, the duration of the experi- ence is extracted. Unfortunately when closely investigating the date recognition capabilities of NER modules, it can be seen, that too many date formats are left unrecognized.
01. 12. 2012 - Present Currently Masters Dissertation 01.12.2012 - Present Currently Masters Dissertation Dec, 2012 to 31 Mar, 2013 Responsibilities 01/2017-05/2017 Motius GmbH
July 2009 - August 2009 bald eagle Aug. 2015 - Nov. 2016 Planet Beach 03/2017 - 11/2017 Motius GmbH
03.17 - 11.17 Motius GmbH

The output of both libraries are displayed on Figure 14 and on Figure 15 correspond- ingly. The results show that a minimum of half the dates were incorrectly identified at both libraries, moreover Spacy even made the error of labeling a date as a Geopolitical Entity (GPE), thus making duration extraction almost impossible.
Since the try and use a NER module for date recognition was proven unsuccessful, the duration is extracted employing a series of Regexp searches on the experience's text.

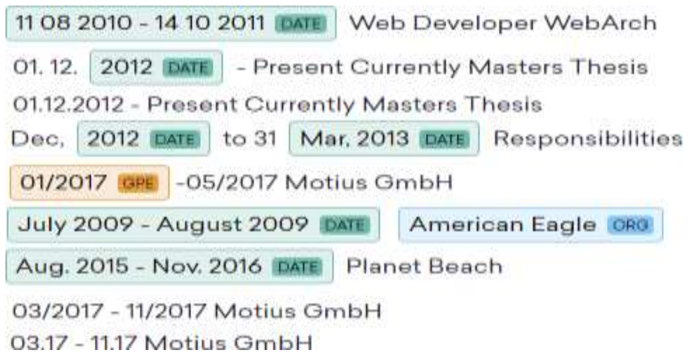

11 08 2010 - 14 10 2011 DATE Web Developer WebArch
01. 12. 2012 DATE - Present Currently Masters Thesis
01.12.2012 - Present Currently Masters Thesis
Dec, 2012 DATE to 31 Mar, 2013 DATE Responsibilities
01/2017 GPE -05/2017 Motius GmbH
July 2009 - August 2009 DATE American Eagle ORG
Aug. 2015 - Nov. 2016 DATE Planet Beach
03/2017 - 11/2017 Motius GmbH
03.17 - 11.17 Motius GmbH

Figure 6: Date recognition using the NER module of Spacy

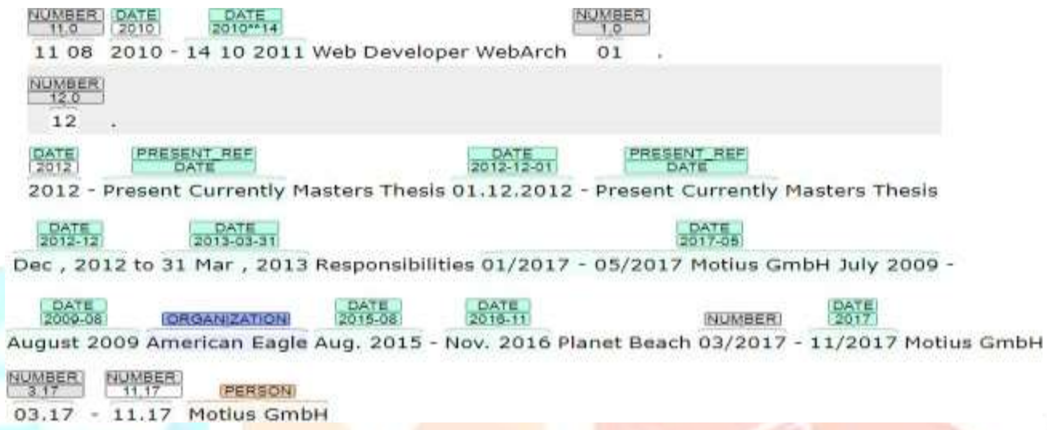

NUMBER 11.0 DATE 2010 DATE 2010**14 NUMBER 1.0
11 08 2010 - 14 10 2011 Web Developer WebArch 01 .
NUMBER 12.0
12 .
DATE 2012 PRESENT_REF DATE DATE 2012-12-01 PRESENT_REF DATE
2012 - Present Currently Masters Thesis 01.12.2012 - Present Currently Masters Thesis
DATE 2012-12 DATE 2013-03-31 DATE 2017-05
Dec , 2012 to 31 Mar , 2013 Responsibilities 01/2017 - 05/2017 Motius GmbH July 2009 -
DATE 2009-08 ORGANIZATION DATE 2015-08 DATE 2016-11 NUMBER DATE 2017
August 2009 American Eagle Aug. 2015 - Nov. 2016 Planet Beach 03/2017 - 11/2017 Motius GmbH
NUMBER 3.17 NUMBER 11.17 PERSON
03.17 - 11.17 Motius GmbH

Figure 7: Date recognition using the NER modules of CoreNLP

Writing one Regexp for date extraction isn't complex nor difficult, but with the rising number of supported date formats the complexity of those expressions steadily grow. as an example a Regexp that searches for the date of format "MM/YYYY" must not match the month and year of dates that were written within the "DD/MM/YYYY" format, whether or not these partially contain the "MM/YYYY" format. the requirement for such mutual exclusion among all Regexps makes the event significantly tougher.
The current implementation support the extraction of the subsequent duration of the format:
• Closed date zero in "MM/YYY" format using numeric months (04 2013 - 05 2014)
• Closed date target "MM/YYY" format using alphabetic months (April 2013 - May 2014)
• Closed date aim "MM-MM/YYY" format using numeric months (03-05 2013)
• The month is written employing a leading zero (05 or 5)
• The month is written using an abbreviations or not (Aug. or August)

Better work experience identification

The implementation within the previous section was supported a basic assumption: a brand new work experience was assumed on every occasion, when the a given line had an increased horizontal distance from the previous line and if the left-margin of the present line was smaller than the left-margin of the previous line. Since such visual styling is is sort of natural to humans, this approach worked ca. 70% of the time. However in the opposite 30% of the cases, people used a special visual styling to differentiate work experiences.

Finding sections during a resume was a better task, as sections were usually indicated by known keywords (e.g.: work experience, education, skills etc.). Unfortunately just in case of finding individual work experiences no such universal keywords exist. However as most of the individual work experiences are absolute to a given duration (e.g.: 15/12/2017-15/01/2018), the quantity of such time intervals are often wont to check, whether the work experience identifier algorithm has found the proper number of experiences. for instance, if the whole work experience text contains five time intervals, but the algorithm later only finds two separate work experiences, it can concluded that the currently used algorithm isn't suitable for the given text, hence another algorithm has got to be used, that hopefully finds all the five work experiences.
To find the quantity of your time intervals Regexps were used. Meanwhile dates are written in numerous formats, the majority of them contain a year written within the "YYYY" format, thus the resulting Regexp focuses on finding years, where the subsequent ten characters don't contain digits. this fashion time intervals, that contain two years (DD/MM/YYYY
- DD/MM/YYYY) are counted just once, since the primary ten characters, that follow the primary "YYYY" contain digits, meanwhile the ten characters, that follow the second "YYYY" don't contain any digits. this manner "open ended" intervals (e.g: "MM/YYYY - "), furthermore as single dates ("DD/MM/YYYY") are counted.
Once the amount of your time intervals is given, an iterative approach is employed to search out the correct work experience extraction algorithm. If the quantity of extracted work experiences is a smaller amount than half the amount of your time intervals, the following work experience extraction algorithm is employed within the line.

Skill recognition

During skill extraction usually a group of known skills (stored in an exceedingly dictionary) are matched against the input text. However if the programming languages have short names (e.g.: "c" or "r"), then simply trying to find occurrences of those won't work, as

such short expressions are going to be matched within the names of various tools. for instance, the programing language "r" is matched within the word "Android".

On the primary glance this problem is solved using tokenization: the input text is tokenized then the tokens of the text are matched against the skill dictionary. For this purpose one can use a (modified) tokenizer from Chapter 3, or one can write his own tokenizer. Either way, there's a significant problem with this approach, as skill expressions, that contain multiple words (e.g.: Microsoft Office, Selenium Webdriver etc.) won't be picked up, since these expressions are tokenized into two separate words.
In order to incorporate skills that are a mix of multiple words, the subsequent approach was used:
• As the extracted text often contains exotic Unicode characters (geometrical shapes, that were used for marking bullet points), or characters from the private Unicode area1, the text is first cleared from the such unique characters.
• Then for every skill within the dictionary, the skill is inserted into a complicated Regexp, that uses the design behind and lookahead2 features of the language to search out a match within the input text. This makes sure that a skill is simply matched within the text, if the character before and after the word are non-alphanumeric, hence the skill "r" won't be matched within the word "Android", however skills containing multiple words are still supported.
The above described process works well, however there's a minor issue with words, that partially contain names of other skills. as an example, if the dictionary contains the words "visual studio 2017" and "visual studio" and therefore the skill section contains a line with the skill "visual studio 2017", to induce eliminate this effect the subsequent steps were used:
• First the dictionary is sorted supported the entries' length in decreasing order.
• The program takes each skill from the list (starting from the longest) and appears for occurrences within the text. Once found it appends the skill to the list of found skills, and deletes the occurrence from the initial text. A disadvantage of the approach is that the list must be more experienced sequentially, hence no for-loop parallelization can happen to hurry up the method. the entire implementation of the skill recognizer algorithm is shown on Listing 5.4.

Listing 1: Implementation of the skill finding algorithm

```
skills = resource_string(
'resources.Extracted.lists', "skills_to_find.txt").decode(
"utf-8", "strict").splitlines()
skills.sort(key=len, reverse=True) found_skills = []
for skill in  skills:
match = re.search(r'((?<=^)|(?<=[^a-zA-Z\d]))' +
re.escape(skill.lower()) + r'(?=$|[^a-zA-Z\d])', text.lower()) if match:
text = text.replace(match.group(), "") found_skills.append(skill)
return found_skills
```

The initial dictionary, that was used for this measurement contained StackOverflow's top 1000 tags^1. At the end of the enhancement a measurement was conducted in order to find out the performance of the newly written skill recognizer. The achieved precision was 58.87%, while the recall was 55.32%. The results are far from perfect, thus a few extra measures were taken in order to achieve a better performance.

Cleaning the dictionary

The initial dictionary that was used had two majors flaws: • If a skill consisted of multiple words, the words were separated by a hyphen ("-") rather than a area. • The dictionary also contained lots of meaningless words, like "testing" or "performance", that in itself didn't describe the persons abilities well. The dictionary was cleared from these words. After cleaning the dictionary contained ca. 650 words. The precision rose to 78%, while the recall rose to 73%. This shows that the standard of the used dictionary incorporates a major impact on the parser's performance.

Learning yet unknown skills

The skill recognizer still lacked a great functionality, namely it was unable to extend its own dictionary. In the skill section people tend to list skills after each other, thus it makes sense to assume that a given word is a skill, if both its neighboring words are also (known) skills. Thus the idea is to tokenize the text and then check whether a given token's neighbors are skills. The steps of the learning process are the following:

- The skill section is cleaned, so that it doesn't contain any special Unicode characters. End of sentence punctuation are also replaced by a blank space. At this point the text could tokenized by an external library, however as the cleaned text is nothing but tokens separated by blank spaces, it makes more sense to simply split the text along these, and thus reduce the dependency on external libraries.

- The resulting list of tokens are cleared from the most common English words, from the so-called "stop words"^{1}. For example the words "and", "to", "him", "they", "but", "if" are stop words. This step was entirely delegated to NLTK.

- Each token is analysed, whether it is a known skill (i.e. it was already in the predefined dictionary) or it is an unknown word.

- If a word is identified as unknown, however its neighbouring words were previously identified as skills, it is assumed that the word is a new skill, and it is added to the dictionary. To make this step work effectively, the stop words had to be removed in the previous steps, as if people use a phrase such as "I know C# and C++", the word "and" would have been picked up as a skill.

The above described learning process works well, however it tends to pick up rather irrelevant words. For example once the sentence "I have mastered the C# language and C++" has been tokenized and cleared from stop words, the word "language" is surrounded by two programming languages ("C#", "C++"), and hence it is identified as a new skill. In order to avoid littering the (once already cleaned) dictionary with such words, a new dictionary was introduced, which contains words, that should be ignored by the skill learning algorithm. Thus in the learning process only skills are added to the dictionary that are not on the list of ignored skills.

Using the 25 resumes of the test set, 11 new skills were added to the dictionary. Once the skill dictionary has been updated with the newly learned words, the skill section is looked through for occurrences of skill words, just as it was done in the already existing solution.

Profiling

For measuring each step's execution time, Pycharm's profiler was used, just as in the previous iteration. The results shown on Table 10.

Table 1: Profiling at the end of the second iterations

Task	Time (s)	Time (%)
Convert PDF to a HTML	14.29	6.2
Convert each line in the HTML re-sume to an object	192.99	83.8
Analyzing keywords and their visual properties	1.9	0.8
Break resume into sections	0.87	0.4
Find individual experiences	0.2	0.6
Find duration in experiences	2.531	1.1
Find skills in experiences	12.8	5.6
Find skills in skill text	4.11	1.8
Write results to file	0.022	0.0

As seen on the results the process for searching skills had a significant impact on the performance. The program has made ca. 450'000 calls to the search function of the Regexp module, while searching for skills.

Compared to the previous version, the total time spent on converting the resume to HTML and to an internal object, accounts for 90% of the total time, while the remaining 10% is spent on finding skills (7.6%), extracting dates (1.1%) and breaking up the work experience section into individual experiences using the more complex algorithm (0.6%). While the main bottle neck is still the conversion process, it would be worth to investigate, whether it is possible to write a more efficient skill recognizer.

Accuracy & Recall

Regarding the skills, the measurement has been extended with "skill recognition" section, which focuses on evaluating to the accuracy and recall of recognized skills. Unlike at the "skill text", a meaning can be given to the term "false positive", namely if a skill is unwanted, hence the precision value is calculated accordingly to its textbook definition.
By having the skill learning algorithm, the recall value of the "skills recognized" is expected to rise. The results are shown on Table 11.

Table 2: Accuracy and recall at the end of the second iteration

Name	Precision (%)	Recall (%)
start date	88	84.6
end date	91.6	81.4
description	86.1	96.5
skill text	1	87.9
skills recognized	78.2	76.2

As expected, the precision and recall values sharply rose for both start and end-date extraction. The precision of work experience description also rose with 10%, however its recall only rose with 2%. The reason for this lies in the details of the algorithms that does the evaluation. Then the evaluation algorithm tries to find the text of each (original) work experience, in the output. Since the output contains all the texts combined, it finds these successfully, hence the high recall. However, as the output is a mixture of all the original work experiences, the precisions are penalized, which means that the output after the second iteration is not 100% clean and mixed work experiences are not uncommon. The current date recognition algorithm could be still enhanced, by supporting more date formats, such as the "YYYY/MM" format, or dates where the year has been abbreviated ("YY/MM"). With these two enhancements the precision could be pushed even higher.
Finally the values for the skill recognition (as expected) also rose, compared to the measured values in Section 5.3.2.2.

4. Conclusion

As seen, some commercial parsers also have 60-70% recall values, while the best open-source parser has recall values scattered int he range of 25-60%. On the other hand the precision values of the parser are slightly lower than the values of commercial and open-source parsers. The main cause for this is bad or inconsistent conversion of PDF to HTML, thus it is an additional reason to replace the information extraction module of the parser. The date recognition algorithm works well, however it doesn't support all the date formats, and this is also reflected in the lower precision values. In order to compete with the commercial parsers, the parser should be able to recognize a much larger variety of dates.
Finally it is worth to note that both the recall and precision values are far from Rapid- parser's values, which are above 90%, thus there is definitely room for improvement.

5. References

Eysenck, M. W. (2006). *Fundamentals of cognition*. Psychology Press.

Gamma, E. (1995). *Design patterns: elements of reusable object-oriented software*. Pearson Education India.

Gil, R. (2015). Introduction to web scraping. Retrieved January 5, 2018, from http: //ruipgil.com/2015/12/17/introduction-to-web-scraping.html

Goldberg, Y. (2016). A Primer on Neural Network Models for Natural Language Processing. *J. Artif. Intell. Res.(JAIR)*, *57*, 345–420.

Gorkovenko, M. (2017). Scraping a JS-Rendered Page. Retrieved January 5, 2018, from http:// stanford. edu/~mgorkove/ cgi- bin/ rpython_ tutorials/ Scraping_ a_ Webpage_Rendered_by_Javascript_Using_Python.php

Hardeniya, N. (2015). *NLTK essentials*. Packt Publishing Ltd.

He, Y. & Kayaalp, M. (2006). A Comparison of 13 Tokenizers on MEDLINE. *Bethesda, MD: The Lister Hill National Center for Biomedical Communications*, 48.

Hop, A. (2016). How to Scrape Javascript Rendered Websites with Python &Selenium. Retrieved January 5, 2018, from https://medium.com/@hoppy/how-to-test-or- scrape-javascript-rendered-websites-with-python-selenium-a-beginner-step-by- c137892216aa

Jones, K. S. (1994). Natural language processing: a historical review. In *Current issues in computational linguistics: in honour of Don Walker* (pp. 3–16). Springer.

Jurish, B. & Würzner, K.-M. (2013). Word and Sentence Tokenization with Hidden Markov Models. *JLCL*, *28*(2), 61–83.

Kaur, A. & Chopra, D. (2016). Comparison of text mining tools. In *Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on* (pp. 186–192). IEEE.

Kibble, R. (2013). Introduction to natural language processing.

Kiss, T. & Strunk, J. (2006). Unsupervised multilingual sentence boundary detection. *Computational Linguistics*, *32*(4), 485–525.

López, F. & Romero, V. (2014). *Mastering Python Regular Expressions*. Packt Publishing Ltd.

Manning, C. D., Schütze, H. et al. (1999). *Foundations of statistical natural language processing*. MIT Press.

Màrquez, L., Padro, L., & Rodriguez, H. (2000). A machine learning approach to POS tagging. *Machine Learning*, *39*(1), 59–91.

Meystre, S. M. & Haug, P. J. (2005). Comparing natural language processing tools to extract medical problems from narrative text. In *AMIA annual symposium proceedings* (Vol. 2005, p. 525). American Medical Informatics Association.