University of Stavanger

Faculty of Science and Technology

MASTER'S THESIS

Study program/ Specialization: Master of Science in Computer Science	Spring semester, 2015 Restricted access
Writer: Thomas Larsson Kleveland	………………………………………… (Writer's signature)
Faculty supervisor: Reggie Davidrajuh External supervisor(s): Derek Göbel	
Thesis title: The application of fuzzy text recognition and -manipulation technologies to clean-up, idealize, improve, and integrate sets of unstructured data	
Credits (ECTS): 30 ECTS	
Key words: Text processing Entity Resolution Fuzzy matching	Pages: 42 + enclosure: 2 Stavanger, 15th of June, 2015. Date/year

The application of fuzzy text recognition and -manipulation technologies to clean-up, idealize, improve, and integrate sets of unstructured data.

Thomas Larsson Kleveland
Computer Science
University of Stavanger
15th of June, 2015

Table of Contents

Table of Figures

Acknowledgements

This thesis is the end product of my study of Master of Science in Computer Science. I want to thank both my faculty supervisor, Reggie Davidrajuh, and my external supervisor, Derek Göbel, for great guidance and advice along the way. I also want to thank my colleague, Paolo Predonzani, for his technical expertise.

Thomas Larsson Kleveland, Stavanger, 15th of June, 2015

1 Introduction

The topic area of this master thesis research is around integrated teams in the Oil & Gas Exploration & Production area. Many small to medium size initiatives in Oil & Gas companies are managed by teams of 10 to 50 people from different disciplines. They each contribute from their own area of work and expertise. These are many, ranging from activity scheduling, resource planning, materials handling, logistics, maintenance engineering and planning, project engineering, to production technology, process engineering, etc. When organized into a project team, their contributions need to be effectively combined to produce a desired outcome on offshore production facilities, drilling rigs or onshore plants and offices. This can be a maintenance program, a drilling program, construction- or other project work, a shutdown/turnaround, everyday operations, or office based projects such as IT projects and real-time data systems.

It is difficult and complex to coordinate the efforts of 10 to 50 people from different disciplines in detail. Many mistakes happen as a result of this complexity, and money is wasted. Therefore in the oil industry, much emphasis is placed these days on integrating their information. Knowledge management systems are an example of such efforts.

We will be developing a solution together with an International Oil & Gas company here in Stavanger that seeks to go 1 step further. The solution will be open to teams like mentioned above. Fuzzy text recognition and -manipulation techniques will be applied to interpret the data from the team members (a lot of which is in spreadsheets), clean it up, idealize it, normalize it, give it meaning, and integrate it such that problems and defects can be spotted at a much greater detail level than can be today, and other interesting things can be done with it.

1.1 Description

In the oil & gas industry, to achieve things, many disciplines are required to collaborate. Between disciplines, the templates and applications that people use, and even the language, can vary. Effective collaboration requires effective information sharing. To achieve this at a small expense and without the need to spend many hours, we need to find a way to automatically improve the data that individuals use in such a way, that end-to-end integration and comparison is made possible.

To this end, this research aims to apply fuzzy text technology such that nomenclature, terminology, abbreviations, units of measure, naming of people and things, are normalized to ‘standard’ wording. With this, we will be able to combine data more easily, and use it to create charts and reports. In choosing our standard for terminology we will follow the ISO15926 industry standard.

We will be creating a range of fuzzy text solutions, and applying them to live data from an oil company. It is our objective to identify the best approaches to:

- Automatically recognize the data type of the most commonly used data in the context of the oil company.
- Automatically enhance this data to standard terminology.
- Integrate the enhanced, normalized data into the data model for comparison, reporting, simulation, and optimization.

The uncertainty that requires our academic approach is twofold:

- We don’t know whether we will be able to automatically classify the data we receive.
- We don’t know whether we will be able to effectively normalize all data that we receive.

In both cases we expect that we may end up with bits of ‘uninterpreted’, ‘un-normalized’ data in our data set. Our experiments will seek to identify the approach that minimizes such occurrences.

2 Theory and results

“In computer science, approximate string matching (often colloquially referred to as fuzzy string searching) is the technique of finding strings that match a pattern approximately (rather than exactly).” [1]

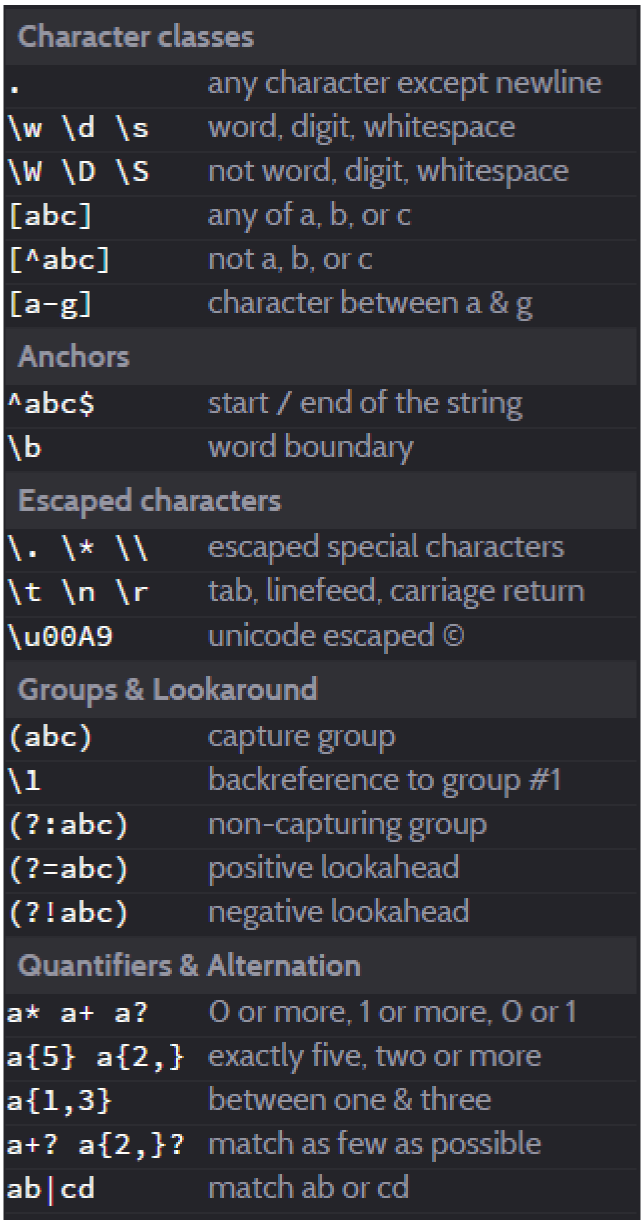
This thesis’ fuzzy text recognition and manipulation is split into five main parts: open text, person names, date/time, currency and general entity recognition. Different strategies has been applied to achieve this goal.

2.1 Open text with regular expressions

“In theoretical computer science and formal language theory, a regular expression (abbreviated regex or regexp and sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. "find and replace"-like operations.” [2]

Learning how to use regular expressions was done with the help of informative websites with explanations and test engines. [3] [4]

Following is a cheat sheet providing an overview of some of the most used regular expression characters:



Character classes	
`.`	any character except newline
`\w \d \s`	word, digit, whitespace
`\W \D \S`	not word, digit, whitespace
`[abc]`	any of a, b, or c
`[^abc]`	not a, b, or c
`[a-g]`	character between a & g
Anchors	
`^abc$`	start / end of the string
`\b`	word boundary
Escaped characters	
`\. \* \\`	escaped special characters
`\t \n \r`	tab, linefeed, carriage return
`\u00A9`	unicode escaped ©
Groups & Lookaround	
`(abc)`	capture group
`\1`	backreference to group #1
`(?:abc)`	non-capturing group
`(?=abc)`	positive lookahead
`(?!abc)`	negative lookahead
Quantifiers & Alternation	
`a* a+ a?`	0 or more, 1 or more, 0 or 1
`a{5} a{2,}`	exactly five, two or more
`a{1,3}`	between one & three
`a+? a{2,}?`	match as few as possible
`ab|cd`	match ab or cd

Figure 1: Regular expression cheat sheet [3]

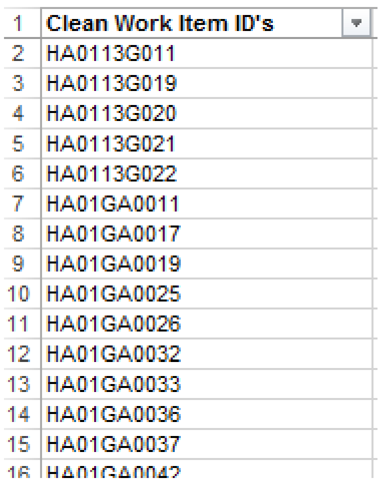
When using a regular expression to search a text with many words sometimes the pattern being searched for can appear inside a word rather than the word itself. For example a regular expression that searches for a number with four digits would find “3700” inside “837003”, even though this is not often desired. By using regular expression lookahead and lookbehind it’s possible to check if the found pattern is not a part of a word. It checks for certain characters at the start and the end of the pattern. Possible characters that marks a start/end of a pattern could be a whitespace (represents either a horizontal or vertical space), comma (for listing), period (end of a sentence) or the end/start of the string.

The regular expression for the lookahead: (?=\s|,|\.|$). It looks for a whitespace, a comma, a period or the end of the string after the pattern that has been found.

The regular expression for the lookbehind: (?<=\s|,|\.|^). It looks for a whitespace, a comma, a period or the start of the string in front of the pattern that has been found. The reason why lookbehind wasn’t listed in the cheat sheet is because regular expressions in JavaScript does not support it.

2.1.1 Regex for Work Item IDs

A spreadsheet of possible Work Item IDs was provided:



1	Clean Work Item ID's
2	HA0113G011
3	HA0113G019
4	HA0113G020
5	HA0113G021
6	HA0113G022
7	HA01GA0011
8	HA01GA0017
9	HA01GA0019
10	HA01GA0025
11	HA01GA0026
12	HA01GA0032
13	HA01GA0033
14	HA01GA0036
15	HA01GA0037
16	HA01GA0042

Figure 2: A part of the Work Item ID spreadsheet

Using this spreadsheet, a pattern was found by examining the different kinds of IDs:

- The IDs always started with one of the characters *H, K, L, P, Q, R* or *S*. The pattern for this behaviour for a regex language is expressed as $[HKLPQRS]$.
- The second character could be any uppercase letter: $[A-Z]$.
- The following *5* characters was either an uppercase letter or a number: $[A-Z\backslash d]\{5\}$.
- The preceding *2* or *3* characters was a number: $\backslash d\{2,3\}$.
- The next character was sometimes a period: $\backslash.?$. As the period already is a regular expression character defining any character except newline, it had to be escaped.
- The remaining characters were always numbers: $\backslash d*$.

This resulted in a first draft of the regular expression:

$$[HKLPQRS][A-Z][A-Z\backslash d]\{5\}\backslash d\{2,3\}\backslash.?\backslash d*$$

However, using this regular expression could sometimes find an ID inside a word. For example "TIHA01GA001956" is not a Work Item ID, but it would find "HA01GA0019" inside the word, which is a Work Item ID. This is a rare case, but is solved, as mentioned earlier, using lookahead and lookbehind.

Combining these additions, the second and final draft for the Work Item IDs was created:

$$(?<=\backslash s|,|\backslash.|\hat{})[HKLPQRS][A-Z][A-Z\backslash d]\{5\}\backslash d\{2,3\}\backslash.?\backslash d*(?=\backslash s|,|\backslash.|\$)$$

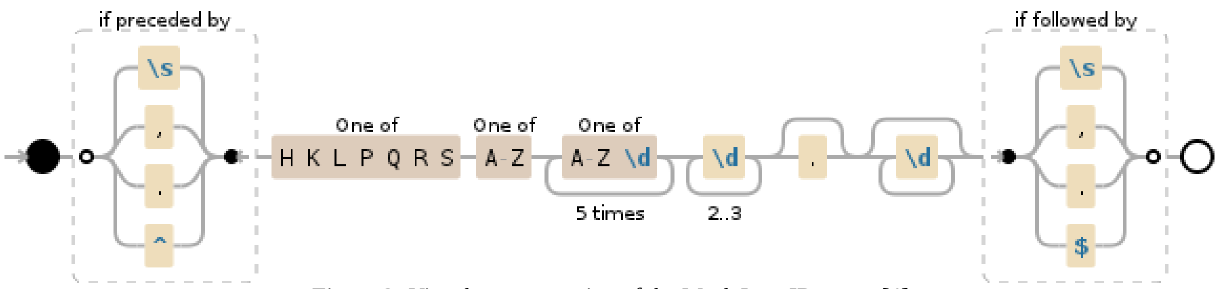
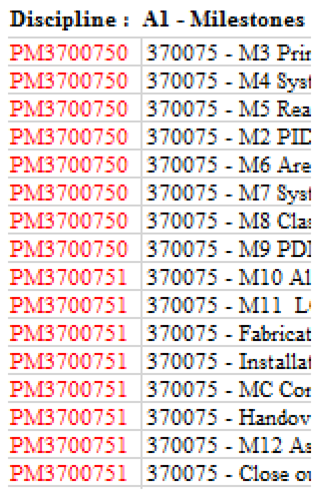


Figure 3: Visual representation of the Work Item ID regex [4]

This regular expression correctly identified all of the Work Item IDs (in red) in a large excel document:

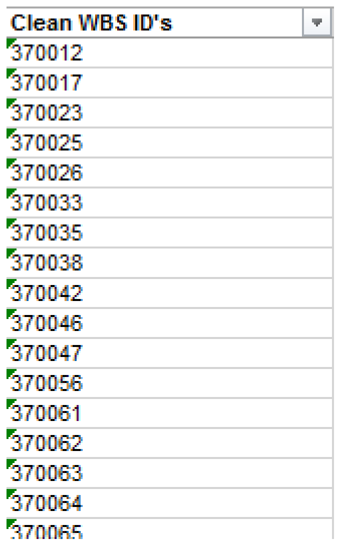


Discipline :	A1 - Milestones
PM3700750	370075 - M3 Prir
PM3700750	370075 - M4 Syst
PM3700750	370075 - M5 Rea
PM3700750	370075 - M2 PIL
PM3700750	370075 - M6 Are
PM3700750	370075 - M7 Syst
PM3700750	370075 - M8 Clas
PM3700750	370075 - M9 PDI
PM3700751	370075 - M10 Al
PM3700751	370075 - M11 L
PM3700751	370075 - Fabricat
PM3700751	370075 - Installat
PM3700751	370075 - MC Cor
PM3700751	370075 - Handov
PM3700751	370075 - M12 As
PM3700751	370075 - Close or

Figure 4: A part of the spreadsheet where Work Item IDs were identified

2.1.2 Regex for WBS (Work Breakdown Structure) IDs

A spreadsheet of possible Work Item IDs was provided:



Clean WBS ID's
370012
370017
370023
370025
370026
370033
370035
370038
370042
370046
370047
370056
370061
370062
370063
370064
370065

Figure 5: A part of the WBS ID spreadsheet

Using this spreadsheet, a pattern was found by examining the different kinds of IDs:

- The WBS IDs always started with the number 3700 .

- The following two characters after that were numbers: $\backslash d\{2\}$.

As a number like *370000* likely can appear inside another number, the same lookbehind and lookahead as for the work item ID's was added:

$$(?<=\backslash s|,|\backslash.|\hat{}\,)3700\backslash d\{2\}(?=\backslash s|,|\backslash.|\$)$$

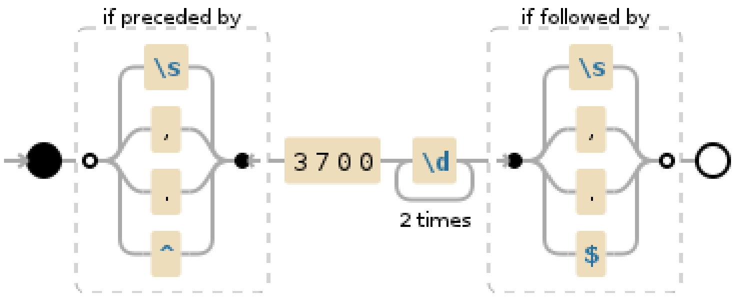
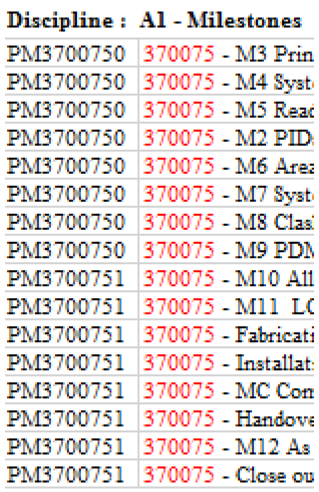


Figure 6: Visual representation of the WBS ID regex [4]

This regular expression correctly identified all of the WBS IDs (in red) in a large excel document:



Discipline :	A1 - Milestones
PM3700750	370075 - M3 Prin
PM3700750	370075 - M4 Syst
PM3700750	370075 - M5 Reac
PM3700750	370075 - M2 PID
PM3700750	370075 - M6 Area
PM3700750	370075 - M7 Syst
PM3700750	370075 - M8 Clas
PM3700750	370075 - M9 PDN
PM3700751	370075 - M10 All
PM3700751	370075 - M11 LC
PM3700751	370075 - Fabricati
PM3700751	370075 - Installat
PM3700751	370075 - MC Con
PM3700751	370075 - Handove
PM3700751	370075 - M12 As
PM3700751	370075 - Close ou

Figure 7: A part of the spreadsheet where WBS IDs were identified

2.1.3 Regex for WBS codes

A document about the WBS code structure and a gazetteer list over different WBS codes were provided. Using these documents, the following information was extracted and deduced:

- The first letter was a of either *C, Z, O, M* or *N*: $[CMNOZ]$.
- The second character was a period: $\backslash.$.
- The *5* following characters was either a uppercase letter or a digit: $[A-Z0-9]\{5\}$. Here the *word*-regex (\w) is not used as it's short for $[A-Za-z0-9\_]$ and includes the character "_" and lowercase letters.
- The next character was a period: $\backslash.$.
- The following letter was either *A, B, C, D, E, F or Z*: $[ABCDEFZ]$.
- The last characters were optional, but it was always *2* letters optionally, separated by period, followed by *1* to *6* letters or digits: $(\backslash.[A-Z]\{2\}(\backslash.[A-Z0-9]\{1,6\})?)?$.

The document about the WBS code structure had some additional information regarding the optional *2* letters, suggesting only a small set of letters could be possible. This information did not seem to be accurate as it existed letters outside this set in the gazetteer list. Therefore, the *2* optional letters were made as generic as possible to support letters outside the mentioned set.

As this pattern rarely is found inside a word, the lookahead and lookbehind was not added. After combining the steps, this was the resulting regular expression that was used:

$$[CMNOZ]\backslash.[A-Z0-9]\{5\}\backslash.[ABCDEFZ](\backslash.[A-Z]\{2\}(\backslash.[A-Z0-9]\{1,6\})?)?$$

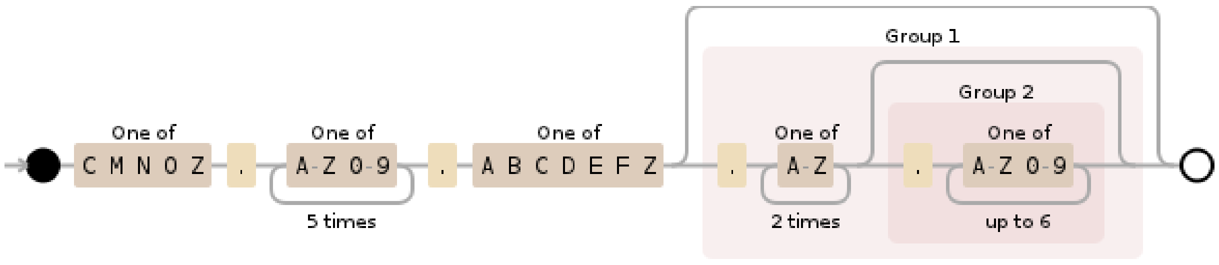


Figure 8: Visual representation of the WBS code regex [4]

This regular expression correctly identified all of the WBS codes in a large excel document.

The regular expression part that supported only the letters mentioned in the WBS code structure document:

$$(CS|D[ADW]|F[EX]|G[EGOS]|I[CNOV]|P[OPR]|RE|S[AE]|TP)$$

2.2 Recognizing a person name in a text

The process of recognizing a person name in a text might seem easy for the human eye as people would be able to spot most person names within a text. However, using a computer to automatically recognize a person name makes things a lot more difficult. It’s hard to define a person name and different approaches were used.

2.2.1 Gazetteer lists with fuzzy matching algorithms

Using this approach the program will test a text against premade gazetteer lists. To be able to support spelling errors, fuzzy matching is needed as well. For testing purposes a spreadsheet is used containing person names and position titles, one entry per cell. The objective is to filter out the person names. As a person name is split into a first name and a last name (and possible more middle names), the full string inside the cell will be split into words. The first word will be tested against the first names gazetteer list, the middle words against the middle names gazetteer list and the last word against the last names gazetteer list. The testing is done by iterating through the list and doing a fuzzy match with the testing word and the gazetteer entry. The fuzzy algorithm used is called Levenshtein distance, which measures the difference between two strings.

```
function LevenshteinDistance(char s[1..m], char t[1..n]):
  // for all i and j, d[i,j] will hold the Levenshtein distance between
  // the first i characters of s and the first j characters of t;
  // note that d has (m+1)*(n+1) values
  declare int d[0..m, 0..n]

  set each element in d to zero

  // source prefixes can be transformed into empty string by
  // dropping all characters
  for i from 1 to m:
      d[i, 0] := i

  // target prefixes can be reached from empty source prefix
  // by inserting every character
  for j from 1 to n:
      d[0, j] := j

  for j from 1 to n:
      for i from 1 to m:
          if s[i] = t[j]:
            d[i, j] := d[i-1, j-1]              // no operation
required
          else:
            d[i, j] := minimum(d[i-1, j] + 1,   // a deletion
                               d[i, j-1] + 1,   // an insertion
                               d[i-1, j-1] + 1) // a substitution

  return d[m, n]
```

Pseudo code 1: Levenshtein distance (Iterative with full matrix) [5]

The distance (or also called edit distance) is measured by counting the number of character edits. A character edit includes insertions, deletions and substitutions. The facts about the Levenshtein distance is largely taken from [5]. As an example, the distance between the word “Bart” and “Berty” is 2. First edit involves changing “a” to “e”, second edit removes the “y”. The normal way of determining if the strings are a fuzzy match is setting a limit of the edit distance. For example setting the max edit distance to be 2 would fuzzy match “Bart” and

"Berty", but not fuzzy match a distance higher than that. This method provides a problem when comparing name parts that has few characters. For example, using a max edit distance of 2, the name "Di" would be fuzzy matched with all names with 2 characters. The solution to this problem is using a normalized version of the Levenshtein distance, where the output of the computation is a number between 0 and 1, where 1 is a perfect match.

$$NormLevDist = 1 - \frac{LevenshteinDistance(origStr,\ gazzStr)}{max(origStr.length,\ gazzStr.length)}$$

Equation 1: Normalized Levenshtein distance

LevenshteinDistance is the method that returns the edit distance, *origStr* and *gazzStr* are the comparing strings.
Using the normalized version, a threshold can be set for fuzzy matching the strings that will work better with all name lengths. As an example, using a threshold of 0.6 would only match the name "Di" with "Di", as more than one edit distance would result in a threshold below 0.5. While the normalized Levenshtein distance between "Bart" and "Berty" would be 0.67, thus resulting in a fuzzy match.

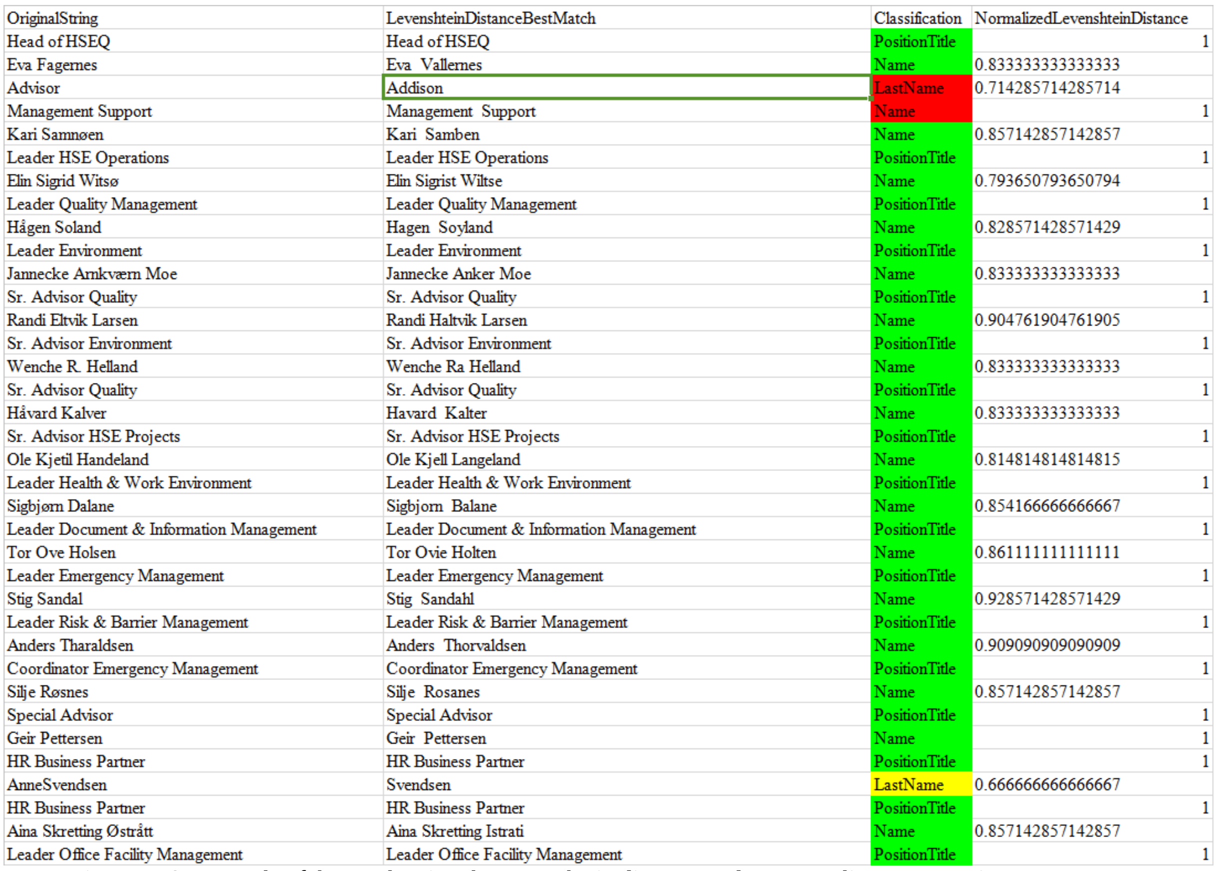
As a person name needed to be tested against different sets of gazetteer lists, an average Levenshtein distance of the combined name will be calculated with the use of the normalized distance of the best matches for each first/middle/last name.

$$AvgBestNormLevDist = \frac{\sum_{i=0}^{n} BestNormLevDist_i}{n}$$

Equation 2: Average Best Normalized Levenshtein distance

Where *BestNormLevDist* is the best normalized distance (closest to 1) a comparison returned when comparing the original string with an entry from the gazetteer list and *n* is number of words in the person name. For example, if the best normalized distance for the first name was 0.91, for the middle name was 0.78 and the last name was 0.86. The average normalized distance of the best matches would be 0.85.

Following picture is an example of the algorithm running through *800+* cells of text testing against gazetteer lists at the size of *100 000+* (last names) and *40 000+* (first names) entries:



OriginalString	LevenshteinDistanceBestMatch	Classification	NormalizedLevenshteinDistance
Head of HSEQ	Head of HSEQ	PositionTitle	1
Eva Fagernes	Eva Vallernes	Name	0.833333333333333
Advisor	Addison	LastName	0.714285714285714
Management Support	Management Support	Name	1
Kari Samnøen	Kari Samben	Name	0.857142857142857
Leader HSE Operations	Leader HSE Operations	PositionTitle	1
Elin Sigrid Witsø	Elin Sigrist Wiltse	Name	0.793650793650794
Leader Quality Management	Leader Quality Management	PositionTitle	1
Hågen Soland	Hagen Soyland	Name	0.828571428571429
Leader Environment	Leader Environment	PositionTitle	1
Jannecke Arnkværn Moe	Jannecke Anker Moe	Name	0.833333333333333
Sr. Advisor Quality	Sr. Advisor Quality	PositionTitle	1
Randi Eltvik Larsen	Randi Haltvik Larsen	Name	0.904761904761905
Sr. Advisor Environment	Sr. Advisor Environment	PositionTitle	1
Wenche R. Helland	Wenche Ra Helland	Name	0.833333333333333
Sr. Advisor Quality	Sr. Advisor Quality	PositionTitle	1
Håvard Kalver	Havard Kalter	Name	0.833333333333333
Sr. Advisor HSE Projects	Sr. Advisor HSE Projects	PositionTitle	1
Ole Kjetil Handeland	Ole Kjell Langeland	Name	0.814814814814815
Leader Health & Work Environment	Leader Health & Work Environment	PositionTitle	1
Sigbjørn Dalane	Sigbjorn Balane	Name	0.854166666666667
Leader Document & Information Management	Leader Document & Information Management	PositionTitle	1
Tor Ove Holsen	Tor Ovie Holten	Name	0.861111111111111
Leader Emergency Management	Leader Emergency Management	PositionTitle	1
Stig Sandal	Stig Sandahl	Name	0.928571428571429
Leader Risk & Barrier Management	Leader Risk & Barrier Management	PositionTitle	1
Anders Tharaldsen	Anders Thorvaldsen	Name	0.909090909090909
Coordinator Emergency Management	Coordinator Emergency Management	PositionTitle	1
Silje Røsnes	Silje Rosanes	Name	0.857142857142857
Special Advisor	Special Advisor	PositionTitle	1
Geir Pettersen	Geir Pettersen	Name	1
HR Business Partner	HR Business Partner	PositionTitle	1
AnneSvendsen	Svendsen	LastName	0.666666666666667
HR Business Partner	HR Business Partner	PositionTitle	1
Aina Skretting Østrått	Aina Skretting Istrati	Name	0.857142857142857
Leader Office Facility Management	Leader Office Facility Management	PositionTitle	1

Figure 9: An example of the result using the Levenshtein distance and gazetteer lists to recognize person names.

The classifications were not that far off, but the best matches rarely matched the person name. The downside of using gazetteer lists is that the result highly depends on the quality of the lists. Splitting up the name and comparing thousands of entries in the gazetteer list with the Levenshtein distance algorithm is computationally expensive. The computation time of the example was around *4* minutes.

2.2.2 Fuzzy searching with Apache Lucene

"Apache Lucene is a free open source information retrieval software library, originally written in Java by Doug Cutting." [6]

Apache Lucene features include text indexing and searching. The search feature includes fuzzy searching. By default, the fuzzy search uses the Damerau-Levenshtein algorithm, but it's also possible to use the classic Levenshtein algorithm. The threshold of the fuzzy search match can be adjusted in the interval between 0 and 1, where 1 is a perfect match. Facts in this paragraph was taken from a website [7].

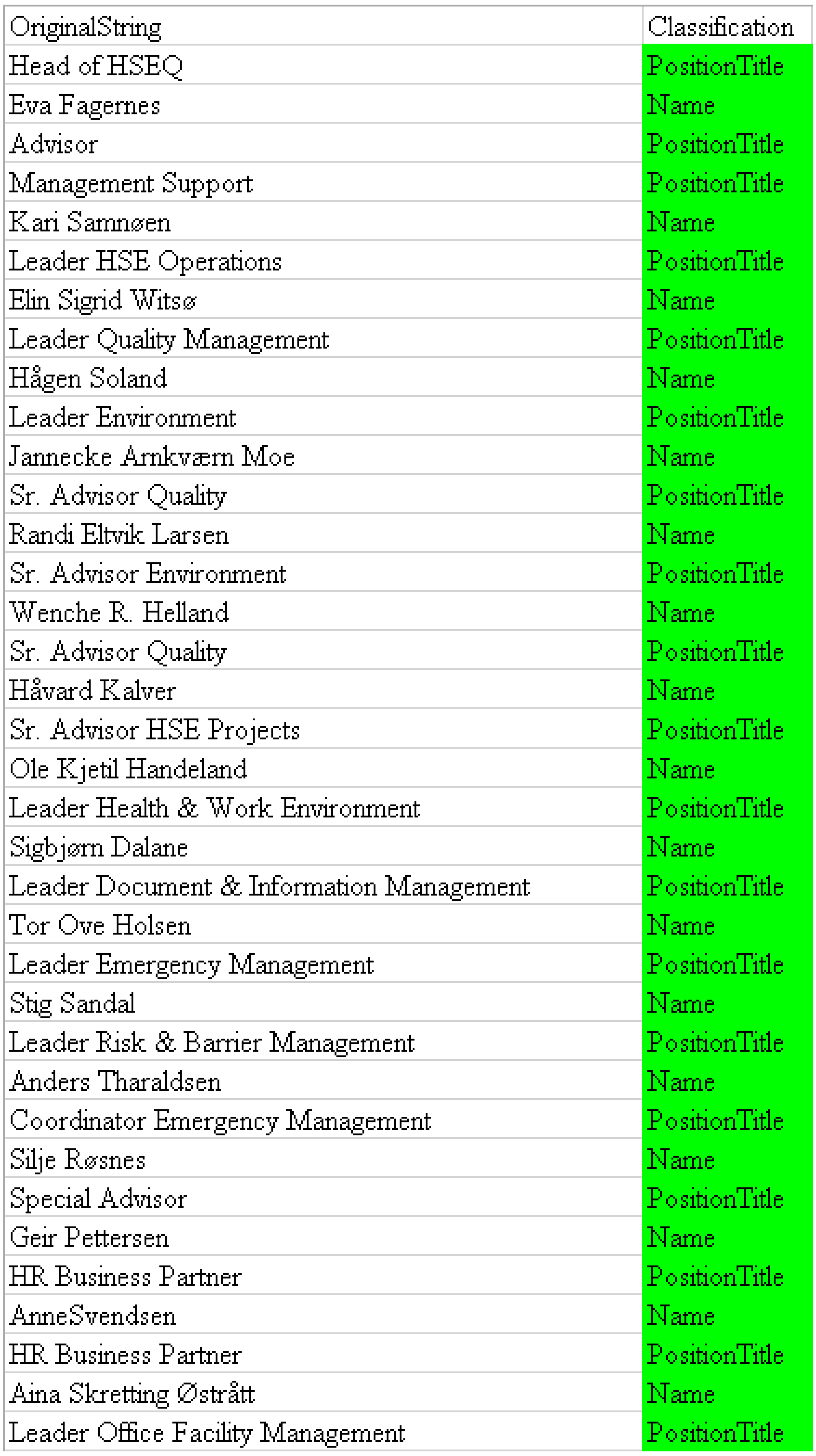
To be able to search a gazetteer list, the list will first need to be indexed. After indexing the gazetteer lists it's possible to run a search towards these indexed lists and the engine will return which list contains the word being searched for. In other words, it could return multiple classifications and it does not say which one of the classifications that are more likely the best one.

2.2.3 Stanford NER (Named Entity Recognizer)

"Stanford NER is a Java implementation of a Named Entity Recognizer. Named Entity Recognition (NER) labels sequences of words in a text which are the names of things, such as person and company names, or gene and protein names. It comes with well-engineered feature extractors for Named Entity Recognition, and many options for defining feature extractors. Included with the download are good named entity recognizers for English, particularly for the 3 classes (PERSON, ORGANIZATION, LOCATION)..." [8]

The Stanford NER implementation is a powerful tool for recognizing person names in a text. It uses a classification dictionary containing persons, organisations and locations to label words. An example of an output of the text "My name is Thomas Kleveland.": "My/O name/O is/O Thomas/PERSON Kleveland/PERSON./O". Where "O" (means "Other") and "PERSON" are the classification labels.

Following picture is an example of the Stanford NER engine going through *800+* cells of text, the rows with the *PositionTitle*-tags were the cells that were not recognized as person names (the cell was either a position title or a person name):



OriginalString	Classification
Head of HSEQ	PositionTitle
Eva Fagernes	Name
Advisor	PositionTitle
Management Support	PositionTitle
Kari Samnøen	Name
Leader HSE Operations	PositionTitle
Elin Sigrid Witsø	Name
Leader Quality Management	PositionTitle
Hågen Soland	Name
Leader Environment	PositionTitle
Jannecke Arnkværn Moe	Name
Sr. Advisor Quality	PositionTitle
Randi Eltvik Larsen	Name
Sr. Advisor Environment	PositionTitle
Wenche R. Helland	Name
Sr. Advisor Quality	PositionTitle
Håvard Kalver	Name
Sr. Advisor HSE Projects	PositionTitle
Ole Kjetil Handeland	Name
Leader Health & Work Environment	PositionTitle
Sigbjørn Dalane	Name
Leader Document & Information Management	PositionTitle
Tor Ove Holsen	Name
Leader Emergency Management	PositionTitle
Stig Sandal	Name
Leader Risk & Barrier Management	PositionTitle
Anders Tharaldsen	Name
Coordinator Emergency Management	PositionTitle
Silje Røsnes	Name
Special Advisor	PositionTitle
Geir Pettersen	Name
HR Business Partner	PositionTitle
AnneSvendsen	Name
HR Business Partner	PositionTitle
Aina Skretting Østrått	Name
Leader Office Facility Management	PositionTitle

Figure 10: An example result of a Stanford NER test when recognizing person names.

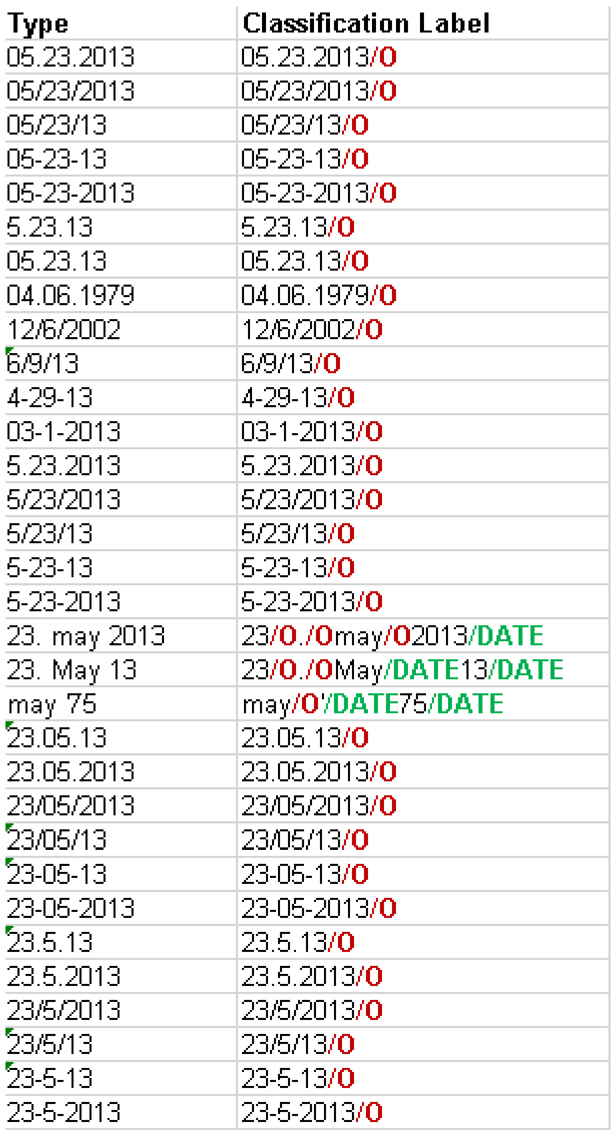
The total computation time for classifying the list was *15* seconds, which was included the seconds it took to load the classification model into the memory. Going through the entire list of *800+* cells, only a few of them were classified wrong. Names with spelling errors were also classified correctly. The downside of classifying the person names using this method is the inability to give good suggestions for potential spelling errors.

2.3 Date recognition and parsing

A date can be in many different formats, several different approaches has been made to figure out the best way to handle date recognition and parsing.

2.3.1 Stanford NER (Named Entity Recognizer)

The Stanford NER also offered a 7-class model, which would label “Time, Location, Organization, Person, Money, Percent, Date”. Following is a test to recognize dates using this model:



Type	Classification Label
05.23.2013	05.23.2013/O
05/23/2013	05/23/2013/O
05/23/13	05/23/13/O
05-23-13	05-23-13/O
05-23-2013	05-23-2013/O
5.23.13	5.23.13/O
05.23.13	05.23.13/O
04.06.1979	04.06.1979/O
12/6/2002	12/6/2002/O
6/9/13	6/9/13/O
4-29-13	4-29-13/O
03-1-2013	03-1-2013/O
5.23.2013	5.23.2013/O
5/23/2013	5/23/2013/O
5/23/13	5/23/13/O
5-23-13	5-23-13/O
5-23-2013	5-23-2013/O
23. may 2013	23/O./Omay/O2013/DATE
23. May 13	23/O./OMay/DATE13/DATE
may 75	may/O'/DATE75/DATE
23.05.13	23.05.13/O
23.05.2013	23.05.2013/O
23/05/2013	23/05/2013/O
23/05/13	23/05/13/O
23-05-13	23-05-13/O
23-05-2013	23-05-2013/O
23.5.13	23.5.13/O
23.5.2013	23.5.2013/O
23/5/2013	23/5/2013/O
23/5/13	23/5/13/O
23-5-13	23-5-13/O
23-5-2013	23-5-2013/O

Figure 11: Stanford NER example output

As the output of the Stanford NER classification failed to label several of the different date formats correctly, another approach was needed.

2.3.2 Using regular expressions

A date can be expressed in several different ways. A regular expression excels at finding text with a certain pattern. The first approach using a regex was to develop a single regex to recognize different patterns of a date in a text with many words that could be something else than dates. Only the final draft will be shown in this chapter, the rest are listed in the enclosure. The regex will be split into different parts and explained individually.

The first part is the day and month. In regular expression language it's not possible to simply express a number range. The range is only for characters. For example $[0-9]$ matches any characters from *0* to *9*. While $[1-31]$ will only be read as characters from *1* to *3* and the character *1* and not the number range *1* to *31*. This makes expressing the number range more complex in a regex:

$$(3[01]|[12][0-9]|0?[1-9])$$

This regex will find any number from 1-31 with an optional 0 in front of single digits. This will be used to identify the *day* part. Generally the date might be followed by a period. In English dates, the day can sometimes be followed by the ordinals *th*, *st*, *nd* or *rd* instead. That is usually also followed up by the word *of* (for example *23rd of May*):

$$(\backslash.?\,|(th|st|nd|rd)\backslash sof)?$$

This type of the date is then followed by the month name.

The regex for the month names:

$$\text{(jan(uary?)?|feb(ruary?)?|mar(s|ch)?|apr(il)?|ma[iy]|jun[ie]?|jul[iy]?|aug(ust)?|sep(t(e mber)?)?|o[kc]t(ober)?|nov(ember)?|de[sc](ember)?)}$$

This regex supports both English and Norwegian month names in full and short form. As many of the month names are equal, the regex is shortened by only having some of the characters different. The combination of the mentioned regexes would support date formats of the type *24th of Oct, 13. May, 06 dec*.

Another way of describing a date is using the 23.12.15 or 9.23.2015 depending on the position of the month number and day number. To be able to support both types of dates, the regex that was used for a day is used for both parts. The symbols separating the day, month and year can be a *period*, a *forward slash* or a *dash*:

$$(3[01]|[12][0-9]|0?[1-9])[-\backslash./](3[01]|[12][0-9]|0?[1-9])$$

The second and last part of the date is the year, limiting this from 1000-2999 or 00-99 for the shorter form:

$$([12][0-9]\{3\}|\backslash d\{2\})$$

To support both uppercase and lowercase month names, the regex is to be used with the flag */i* (ignore case). This can be done in most regex match engines or by adding the flag inside the regex (?*i*) . Adding a lookbehind and lookahead and combining both of the ways of expressing a date, the final regex:

```
(?<=\s|,|^)((((3[01]|[12][0-9]|0?[1-
9])(\.?|(th|st|nd|rd)\sof))?\s?(jan(uary?)?|feb(ruary?)?|mar(s|ch)?|apr(il)?|ma[iy]|jun
[ie]?|jul[iy]?|aug(ust)?|sep(t(ember)?)?|o[kc]t(ober)?|nov(ember)?|de[sc](ember)?)
,?\s?)|(3[01]|[12][0-9]|0?[1-9])[-\./](3[01]|[12][0-9]|0?[1-9])[-\./])([12][0-
9]{3}|\d{2})?(?=\s|,|$)
```

Reading this regex is easier when presented visually:

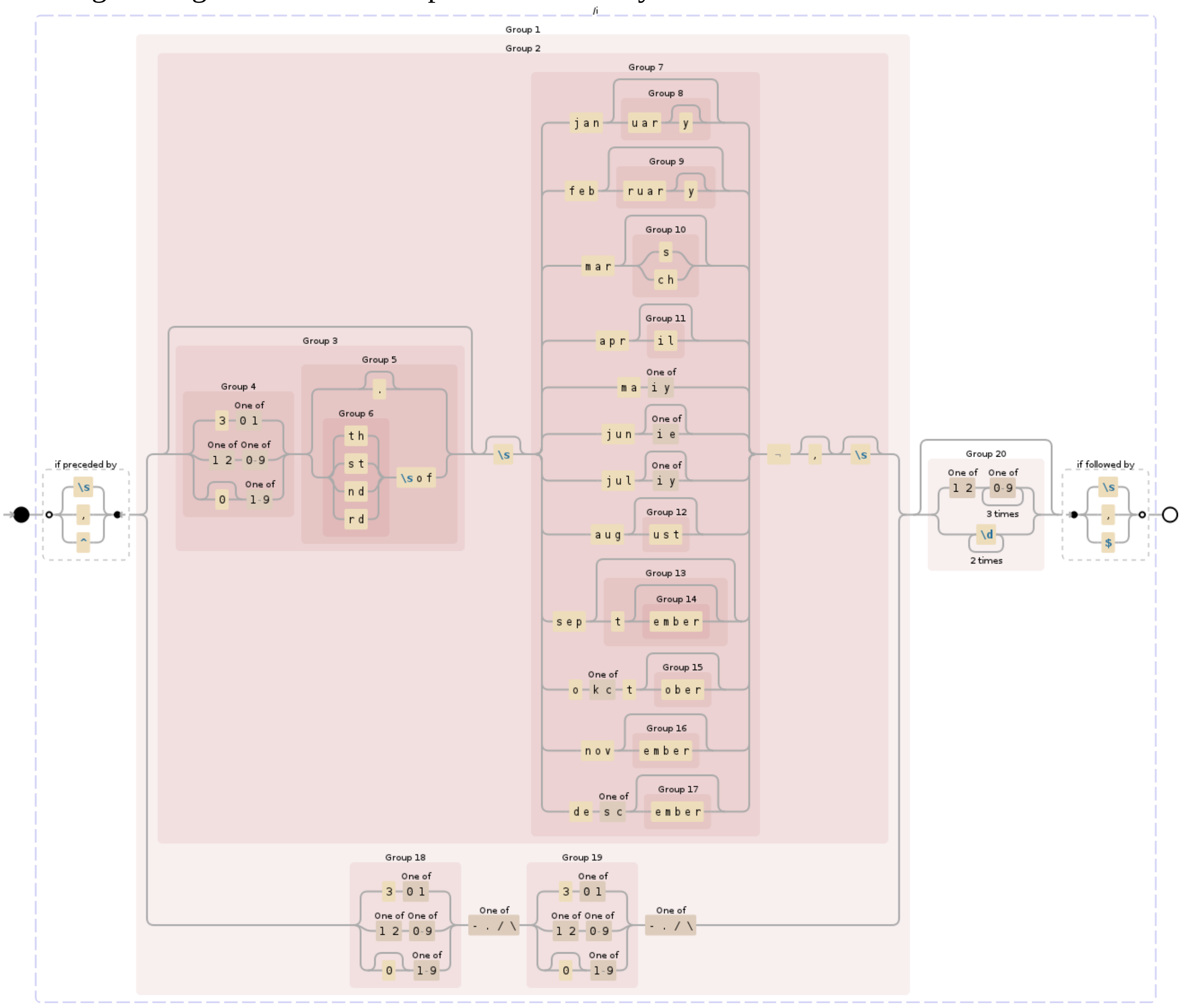
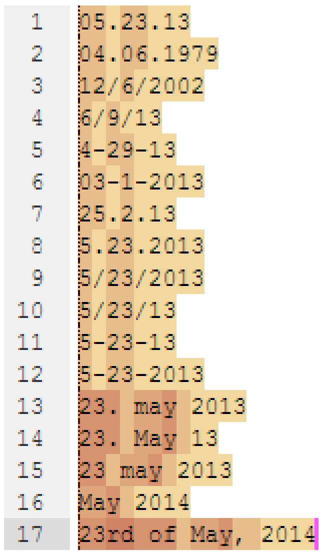


Figure 12: Visual representation of the date recognition regex. Created with Debuggex [4]

The regex supported a lot of different date formats:



```
05.23.13
04.06.1979
12/6/2002
6/9/13
4-29-13
03-1-2013
25.2.13
5.23.2013
5/23/2013
5/23/13
5-23-13
5-23-2013
23. may 2013
23. May 13
23 may 2013
May 2014
23rd of May, 2014
```

Figure 13: Example of supported formats correctly matched with the created regex. Created with Debuggex [4].

It’s hard to create a regex of this magnitude without false positives. An example of a false positive that this regex would find is *30.31.14*. This date will however not successfully parse using a date parser.

The regex did also have false negatives. For example the date *May 23, 2014* would only be matched as *May 23,* as this could mean *May 2023*.

Before looking more into improving the regex, another approach towards recognizing a date was made.

2.3.3 Per cell approach

As the recognition of dates is aimed towards a spreadsheet, another way of looking at it is by a per cell basis. In spreadsheets, a cell can have a type of content, one of the types being date. If a cell is of type date, the date is already recognized. If the cell is of type string, recognizing and parsing potential dates is needed. Instead of creating a complex regex that looks for a complete date, it’s easier to create multiple regexes that matches date parts. An example of a date part could be *October*, *13th* or *2014*. The algorithm will tokenize the string inside the cell into words (split on space) and then iterate through those words and use multiple regexes to test each word if it could be a part of a date. A word in this context is characters separated by a space. If a part of a date is found, a *run* is started and will add the word to a list. The run will continue until a following word is not a part of a date. The preceding words found to be a part of a date will then be attempted to parse. The run will reset and the process will be redone until there are no more words left in the cell.

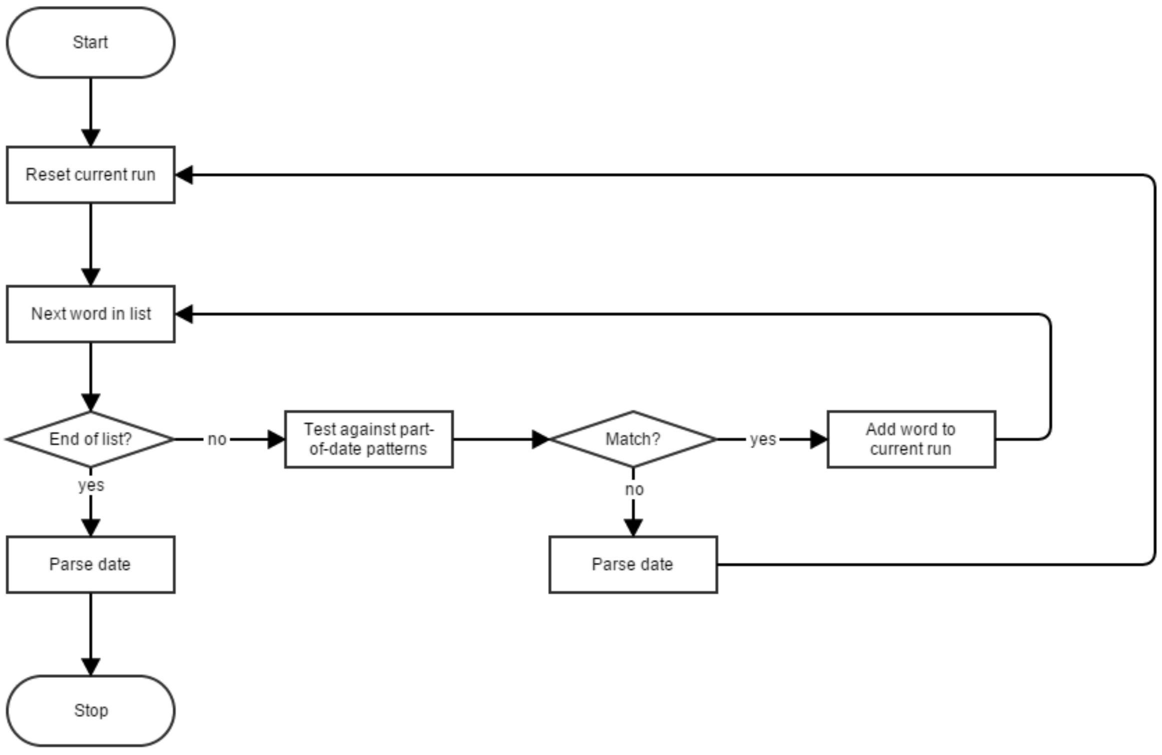


Figure 14: Date recognition flowchart. Created using gliffy [9].

As the words were only tokenized by space, a word could also be a full date, for example *23.05.2014*. A list of type of date parts that should be matched:

- 14
- 78
- 1978
- 23.12.14
- 12.23.2014
- 23-12-14
- 23/12/14
- Any month name
- 14th
- ‘78

The regex for a year, mentioned in the previous chapter, matches the *3* first types:

$$([12][0-9]\{3\}|\backslash d\{2\})$$

This regex will match the 3 first types.

The regex for a day, month number and a year can be expressed using a combination of the day-month regex

$$(3[01]|[12][0-9]|0?[1-9])[-\backslash./](3[01]|[12][0-9]|0?[1-9])$$

And the year regex:

$$(3[01]|[12][0-9]|0?[1-9])[-\backslash./](3[01]|[12][0-9]|0?[1-9])[-\backslash./]([12][0-9]\{3\}|\backslash d\{2\})$$

This regex will match the 4 next types.

The regex for a month name is also mentioned in the previous chapter:

(jan(uary?)?|feb(ruary?)?|mar(s|ch)?|apr(il)?|ma[iy]|jun[ie]?|jul[iy]?|aug(ust)?|sep(t(ember)?)?|o[kc]t(ober)?|nov(ember)?|de[sc](ember)?)

This regex will match the month names.

As there are no difference between *23 May* and *23rd of May*, removing potential ordinals and ignoring the word *of* prepares it better for the parsing process. Removing the ordinals can be done by using a regex to find it:

$$(?<=\backslash d)(th|st|nd|rd)$$

The same goes for *'78* and *78*, where removing the *'* character makes it easier. This makes the total count of *4* regexes for determining whether a word is a part of a date or a full date.

For parsing the date, C# has a method called *DateTime.Parse(string s)* that automatically parses a string using the current locale settings. To use another locale it's possible to use an additional parameter *DateTime.Parse(string s, IFormatProvider provider)*. When a date is not a valid format for the current locale, a *FormatException* is thrown. To be able to support both the date formats where the month number and day number has different places, it's possible to surround the first attempt to parse towards the *en-GB* (the culture code for United Kingdom) locale with a *try/catch*. In the catch clause, the second attempt to parse it towards the *en-US* (the culture code for United States) locale will be made. This will ensure both of the formats to be parsed. However, a problem that did not get handled are the ambiguous dates. For example the date, *06.04.23*, is impossible to determine whether it means *6th of April* or *4th of June*. Further research into solving this could involve looking at other dates in the same spreadsheet.

2.4 Currency recognition

A currency can have many different formats, but the difference between a currency and a normal number is the currency type. The currency types determines the value of the number it is combined with. A number can therefore have different types of value before a specific currency type is associated with the number. The number can also sometimes have multiplying symbols, for example *$1MM* suggests that the number is actually *1 000 000*. The multiplying symbols can sometimes have different meaning, in the oil industry the *M* actually means a million, while the normal abbreviation for a million is *MM*. As the currency notations can vary, the most common notations found in various confidential spreadsheets provided were in following format:

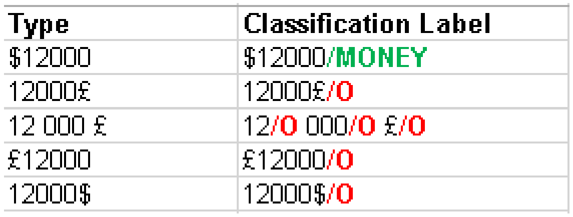
[Number] [Amount Multiplicator]? [Currency code/symbol]
[Currency code/symbol] [Number] [Amount Multiplicator]?

An example of the first format is *200 MNOK*, which is *200 000 000 NOK* in the oil industry.

An example of the second format is *$ 1M*, which is *1 000 000 USD* (or any other currency code with the *$* symbol.

2.4.1 Stanford NER (Named Entity Recognizer)

As mentioned in the chapter regarding date recognition, the Stanford NER had a 7-class model where one of those classes was *Money*. After testing the library against different types of currency formats, only dollars were labelled correctly when it had the form *$12000*:



Type	Classification Label
$12000	$12000/MONEY
12000£	12000£/O
12 000 £	12/O 000/O £/O
£12000	£12000/O
12000$	12000$/O

Figure 15: Stanford NER currency classification

Another approach to recognize currencies was needed.

2.4.2 Using ANNIE with GATE Developer

"General Architecture for Text Engineering or GATE is a Java suite of tools originally developed at the University of Sheffield beginning in 1995 and now used worldwide by a wide community of scientists, companies, teachers and students for many natural language processing tasks, including information extraction in many languages." [10]

By default, the *GATE Developer* program comes with an IE (Information Extraction) system called *ANNIE (A Nearly-New Information Extraction system)*, which uses a combination of different processing resources. The system is developed by *Hamish Cunningham*, *Valentin Tablan*, *Diana Maynard*, *Kalina Bontcheva*, *Marin Dimitrov* and others. It relies on finite state algorithms and the *JAPE* language. The preceding text and the following picture was largely taken from the *GATE* website on the chapter regarding *ANNIE* [11].

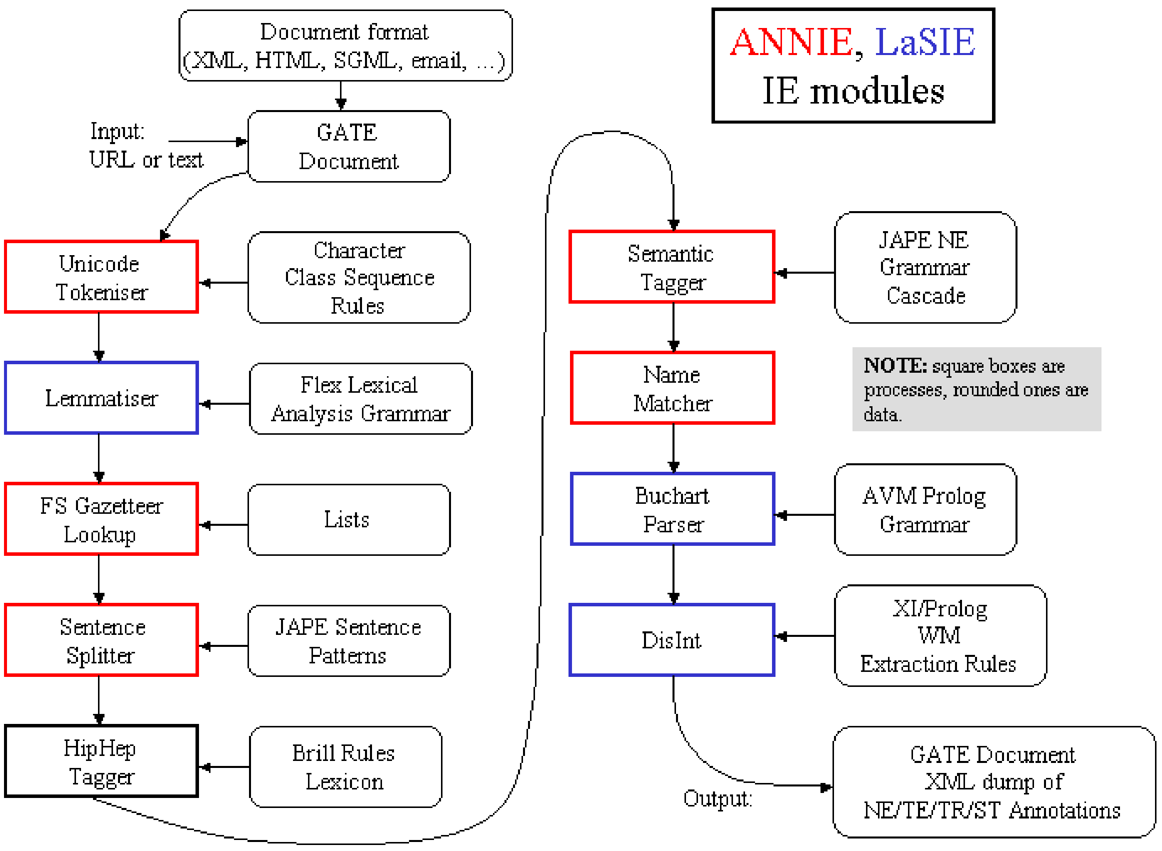
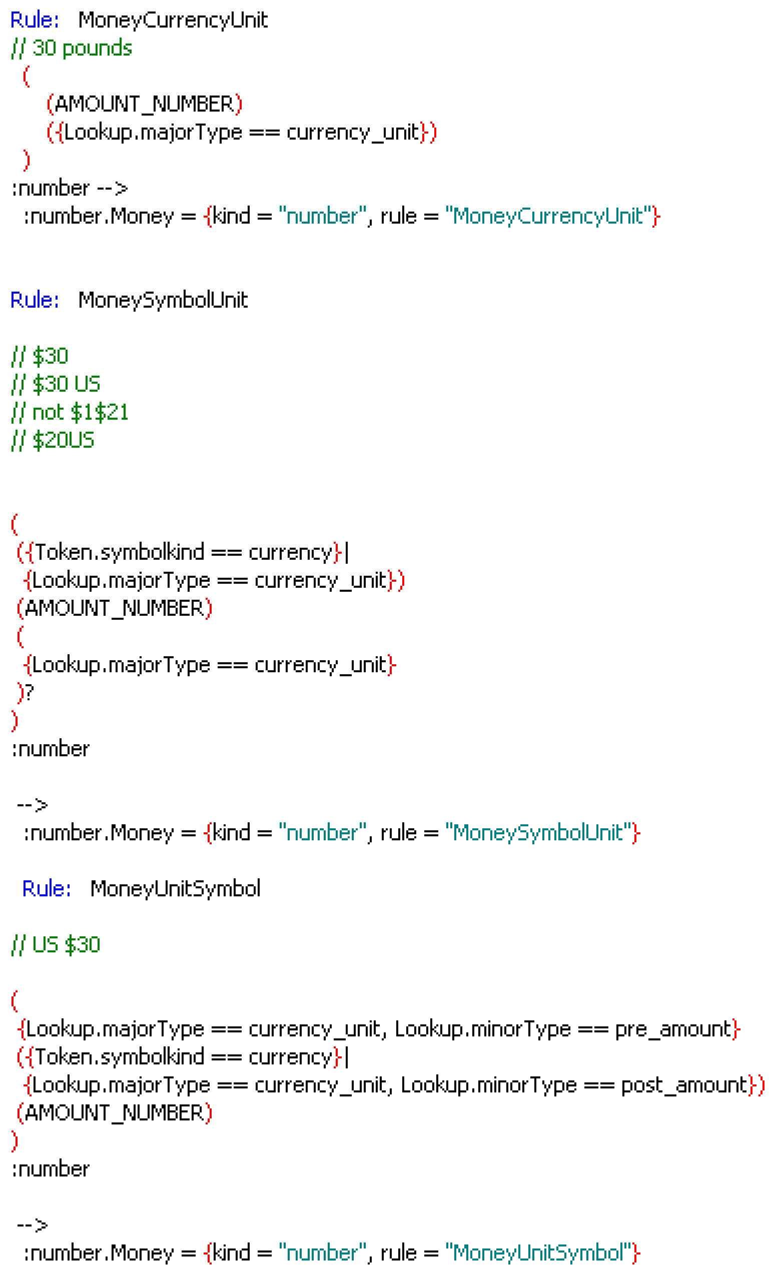


Figure 16: ANNIE components form a pipeline which appears in this figure. [11]

By creating rules for the *Semantic Tagger* component and adding the currency codes/symbols as gazetteer lists, an attempt was made to implement a currency tagger. The rules and macros are created using a language called *JAPE*. The *JAPE* grammar is explained in detail in the *GATE JAPE Grammar Tutorial* [12]. When installing the *GATE Developer* application, several .jape-files were included that served the purpose of tagging different type of text, for example *address*, *email*, *url*, *etc*. One of these types were *number*, which already included a rule for some currency notations:



```
Rule:   MoneyCurrencyUnit
// 30 pounds
 (
     (AMOUNT_NUMBER)
     ({Lookup.majorType == currency_unit})
 )
:number -->
 :number.Money = {kind = "number", rule = "MoneyCurrencyUnit"}


Rule:   MoneySymbolUnit

// $30
// $30 US
// not $1$21
// $20US


(
 ({Token.symbolkind == currency}|
  {Lookup.majorType == currency_unit})
 (AMOUNT_NUMBER)
 (
  {Lookup.majorType == currency_unit}
 )?
)
:number

 -->
  :number.Money = {kind = "number", rule = "MoneySymbolUnit"}

  Rule:   MoneyUnitSymbol

// US $30

(
 {Lookup.majorType == currency_unit, Lookup.minorType == pre_amount}
 ({Token.symbolkind == currency}|
  {Lookup.majorType == currency_unit, Lookup.minorType == post_amount})
 (AMOUNT_NUMBER)
)
:number

 -->
  :number.Money = {kind = "number", rule = "MoneyUnitSymbol"}
```

Figure 17: From the file, number.jape, that came with the application

The rules uses a combination of macros (e.g. *AMOUNT_NUMBER*) and gazetteer (e.g. *Lookup*) lists to tag open text with the appropriate types. The goal was to create *JAPE*-rules that would match the formats mentioned in the introduction of the *Currency recognition* chapter. Looking at the existing rules it seems possible to create such rules for a currency. It looks like it's possible to concatenate "patterns" to match a word, like *$20US* is matched because *$* is a currency symbol (*Token.symbolkind == currency*), *20* is a number (*AMOUNT_NUMBER*) and *US* is a currency unit from the gazetteer list with the *majorType currency_unit*. However, there were some differences in the currency notation format required and the ones that already existed. The difference being that the tokenizer (the program that splits words/sentences into tokens) would split numbers and symbols into separate tokens. Therefore, *$20US* would have *3* tokens (*$*, *20* and *US*), while *MNOK* would just be one token. As it was only possible to handle the tokens individually, the token *MNOK* could not be matched to a gazetteer list of currency codes (because of the prefix *M*). A token could not be tampered with before comparing to a gazetteer list, so there was no way of comparing a token to a combination of two gazetteer lists. For example, it was not possible to compare a token to *{Lookup.majorType = amount_multiplicator} (or referring to a macro)* + {*Lookup.majorType = currency_unit}*, where + symbol meaning they were concatenated.

The implementation of the *JAPE*-rules and gazetteer lists attempts are explained in the implementation chapter 3.3.

With the use of *GATE,* following papers should be cited [13] [14].

2.4.3 Per cell approach

In this chapter, only the theory behind how an approach like this could be made is talked about. There were no implementations or results performed.

Like mentioned in the chapter 2.3.3, a spreadsheet cell can be of certain types, this includes a currency-type. If the cell is of type currency, the currency is already recognized and can be treated as such.

When handling number-formatted cells without a currency code or symbol, it's necessary to look at the context. The currency code can sometimes be in the horizontally adjacent cells or mentioned in a column header if the rows for that column are all numbers. The currency code can also sometimes be mentioned at the start of the document, e.g. *"Numbers in NOK"* or even *"Numbers in $1MM"* which would suggest that the discovered numbers are really *1 000 000* times bigger. If there is a mention of a currency in a header of a table in the spreadsheet, it should be analysed to potentially extract information about other cell's currency codes. For example if a column had the header "MNOK", this would probably mean that the rows for this column should be of type NOK and the actual value is the row value times a million.

When a cell is of type string, the same approach made to recognize a date could be made for a currency, testing each word for the possibility of being part of a currency. Example of how a flowchart of the algorithm would be:

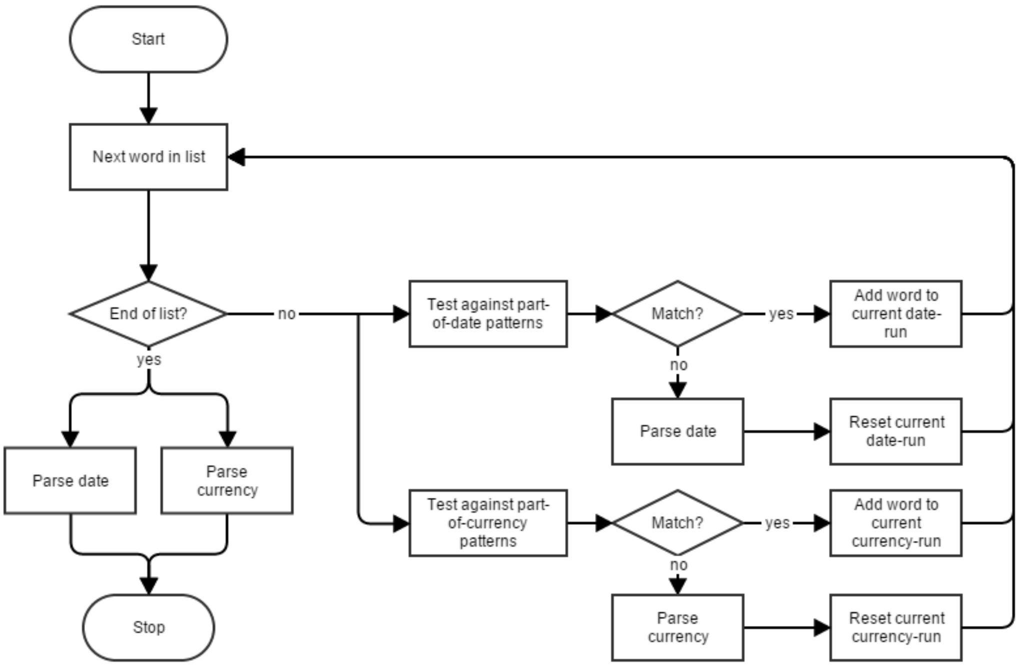

Start
Next word in list
End of list?
no
yes
Parse date
Parse currency
Stop
Test against part-of-date patterns
Match?
yes
Add word to current date-run
no
Parse date
Reset current date-run
Test against part-of-currency patterns
Match?
yes
Add word to current currency-run
no
Parse currency
Reset current currency-run

Figure 18: Currency and date recognition flowchart. Created using gliffy [12].

The currency formats mentioned in the introduction of chapter 2.4 are used to create part-of-currency regexes combined with the use of gazetteer lists of currency symbols and codes.

Part-of-currency examples that should be matched:

- "100,100.10"
- "100"
- "10.100,10"
- "10.100NOK"
- "10.100MNOK"
- "NOK10,000"
- "10.000kr"
- "kr10.000"

The currency formats:

[Number] [Amount Multiplicator]? [Currency code/symbol]
[Currency code/symbol] [Number] [Amount Multiplicator]?

Where *[Number]* is matched using a regex:

```
^[+-]?[0-9]{1,3}(?:[0-9]*(?:[.,][0-9]{1})?|(?:,[0-9]{3})*(?:\.[0-9]{1,2})?|(?:\.[0-9]{3})*(?:,[0-9]{1,2})?)$
```

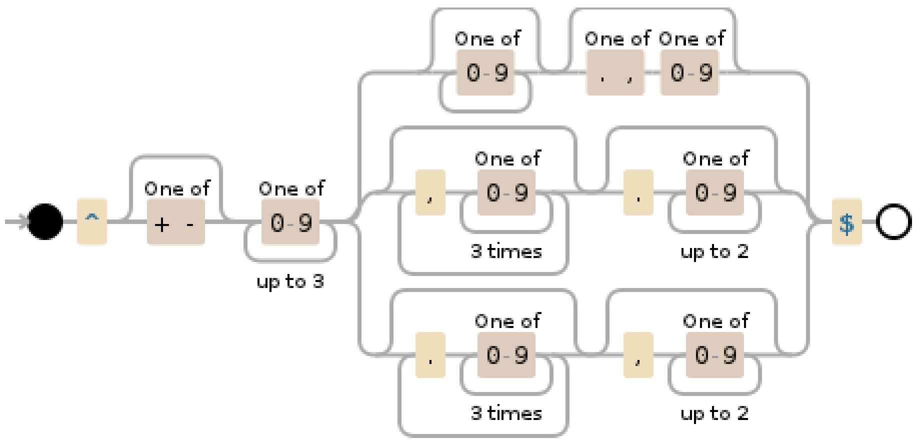


Figure 19: Visual representation of the "Number"-regex

This regex was taken from an answer on stackoverflow.com [15].

The *[Amount Multiplicator]* is matched using a regex:

(?i)[kmb]|mm|[bm]ln|[bm]il[lj]\.?

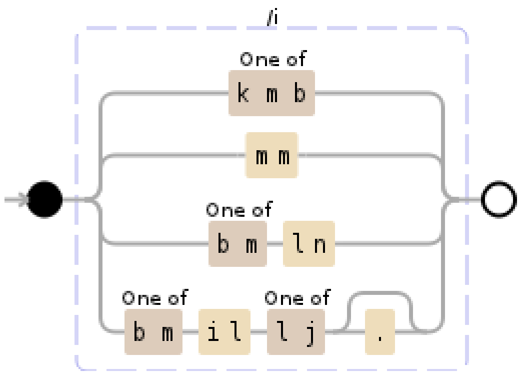


Figure 20: Visual representation of the "Amount Multiplicator"-regex

The *[Currency code]* and *[Currency symbol]* are matched using a gazetteer lists [16] [17].

2.5 General entity recognition using OYSTER (Open sYSTem Entity Resolution)

"Entity Resolution (ER) is the process of determining whether two references to real-world objects are referring to the same object or to different objects." [18]

"The OYSTER (Open sYSTem Entity Resolution) is an entity resolution system that supports probabilistic direct matching, transitive linking, and asserted linking." [19]

There are different types of record linkage. One of them being deterministic record linkage, where two records are said to match if all or some identifiers are identical. And the other one being probabilistic record linkage, also called fuzzy matching, probabilistic merging or fuzzy merging. The method uses thresholds to determine if a pair is a match, a non-match or a possible match. The possible matches can be dealt with by e.g. human review, depending on requirements. This paragraph was largely taken from the Wikipedia page about *Record linkage*. [20]

The foundation for the internal logic of OYSTER is the R-Swoosh algorithm, an algorithm for systematically applying the match and merge functions to arrive at the generic entity resolution of *R (ER(R)),* where *R* is the initial set of entity references. This paragraph was largely taken from the *R-Swoosh Algorithm* part in the book *Entity Resolution and Information Quality* [18].

Using OYSTER, the goal was to discover WBS project title entities in a provided spreadsheet by abiding by certain rules for matching. The rules were as following:

- If the WBS project title is the exact same, it's the same entity.
- If the WBS project title is very close to equal (spelling errors or the use of symbols like "-"), it's the same entity as long as any potential year inside the project title is the same.
- If the WBS project title is cut off, but equal to the start of another string, it's the same entity. For example "High temperature in potable water Cooling sy" is the same entity as "High temperature in potable water Cooling systems".

To be able to match project titles that are very close to equal, a fuzzy match is required. OYSTER comes with multiple fuzzy algorithms. A list of the algorithms with explanations can be found in the OYSTER Reference Guide. These algorithms can be used by specifying them in the "*MatchResult"*-attribute in the "*Attributes*"-file [21]. Several of the algorithms were tested, but the default algorithms offered by OYSTER could not satisfy the mentioned rules for entity matching. A reason for this is that some of the project titles could have a year that would be different, thus being a different entity. Pure fuzzy matching a string where only the year would be different would still result in a match if the threshold is not set to exact match, but fuzzy matching would still be needed for the text that could contain spelling errors or "cut-offs".

To be able to align the entity matching with the rules mentioned, two custom matching algorithms were implemented. Using the custom algorithms, oyster managed to find a total of *51* entities compared to the correct number of *53* of a total *85* input rows.
The reason it did not correctly find *53* entities was because of contradicted matching. E.g.: "*Keep 3D model updated*" was not supposed to be the same entity as "*Keep 3D model updated - 2015*", but "*CAPEX - Installation, Prefab and Estimating TQs for*" was the same entity as "*CAPEX - Installation, Prefab and Estimating TQs for 2014*". This was not possible to create a matching rule for. This caused *2* of the input rows to be wrongly matched to another entity instead of having their own entity created. Comparatively, the closest number of entities created when trying out the predefined OYSTER algorithms was *66* by using the *LED(0.8)*-algorithm. Even though it might've been possible to improve this number by only using the default algorithms, there we no fuzzy algorithm that could take the year- or "cut-off"-problem into consideration. The implementation is described in the chapter 3.4.

3 Implementations

3.1 Apache Lucene

The Apache Lucene implementation and testing was largely done by following a tutorial online [22]. Using the sample code provided on the webpage, the “FileIndexApplication.java”-file was edited to test the engine. Using different gazetteer lists, it was possible to search these indexed gazetteer lists by running:

```
Searcher searcher = new Searcher(INDEX_DIR);
List<IndexItem> result = searcher.findByContent("Managr~0.8",
DEFAULT_RESULT_SIZE);
print(result);
```

This would run a fuzzy search for “*Managr*” with the Damerau-Levenshtein distance threshold of *0.8*.

The printed result:

```
Query: Managr~0.8
----------
PositionTitles.xlsx
0.040840294 = (MATCH) sum of:
  0.040840294 = (MATCH) weight(content:manager^0.8333333 in 2)
[DefaultSimilarity], result of:
    0.040840294 = score(doc=2,freq=26.0), product of:
      0.28785136 = queryWeight, product of:
        0.8333333 = boost
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.48492622 = queryNorm
      0.1418798 = fieldWeight in 2, product of:
        5.0990195 = tf(freq=26.0), with freq of:
          26.0 = termFreq=26.0
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.0390625 = fieldNorm(doc=2)

----------
Lastnames.xlsx
0.009665137 = (MATCH) sum of:
  0.0014967038 = (MATCH) weight(content:manag^0.8 in 1) [DefaultSimilarity],
result of:
    0.0014967038 = score(doc=1,freq=1.0), product of:
      0.54523754 = queryWeight, product of:
        0.8 = boost
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.48492622 = queryNorm
      0.002745049 = fieldWeight in 1, product of:
        1.0 = tf(freq=1.0), with freq of:
          1.0 = termFreq=1.0
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.001953125 = fieldNorm(doc=1)
  0.0046771993 = (MATCH) weight(content:manage^0.8333333 in 1)
[DefaultSimilarity], result of:
    0.0046771993 = score(doc=1,freq=9.0), product of:
      0.56795573 = queryWeight, product of:
        0.8333333 = boost
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.48492622 = queryNorm
      0.008235147 = fieldWeight in 1, product of:
        3.0 = tf(freq=9.0), with freq of:
          9.0 = termFreq=9.0
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.001953125 = fieldNorm(doc=1)
  0.0034912345 = (MATCH) weight(content:manager^0.8333333 in 1)
[DefaultSimilarity], result of:
    0.0034912345 = score(doc=1,freq=76.0), product of:
      0.28785136 = queryWeight, product of:
        0.8333333 = boost
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.48492622 = queryNorm
      0.012128602 = fieldWeight in 1, product of:
        8.717798 = tf(freq=76.0), with freq of:
          76.0 = termFreq=76.0
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.001953125 = fieldNorm(doc=1)

----------
Firstnames.xlsx
0.004126114 = (MATCH) sum of:
  0.001132706 = (MATCH) weight(content:manager^0.8333333 in 0)
[DefaultSimilarity], result of:
    0.001132706 = score(doc=0,freq=2.0), product of:
      0.28785136 = queryWeight, product of:
        0.8333333 = boost
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.48492622 = queryNorm
      0.0039350376 = fieldWeight in 0, product of:
        1.4142135 = tf(freq=2.0), with freq of:
          2.0 = termFreq=2.0
        0.71231794 = idf(docFreq=3, maxDocs=3)
        0.00390625 = fieldNorm(doc=0)
  0.0029934077 = (MATCH) weight(content:manar^0.8 in 0) [DefaultSimilarity],
result of:
    0.0029934077 = score(doc=0,freq=1.0), product of:
      0.54523754 = queryWeight, product of:
        0.8 = boost
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.48492622 = queryNorm
      0.005490098 = fieldWeight in 0, product of:
        1.0 = tf(freq=1.0), with freq of:
          1.0 = termFreq=1.0
        1.4054651 = idf(docFreq=1, maxDocs=3)
        0.00390625 = fieldNorm(doc=0)

Result Size: 3
PositionTitles.xlsx
Lastnames.xlsx
Firstnames.xlsx
```

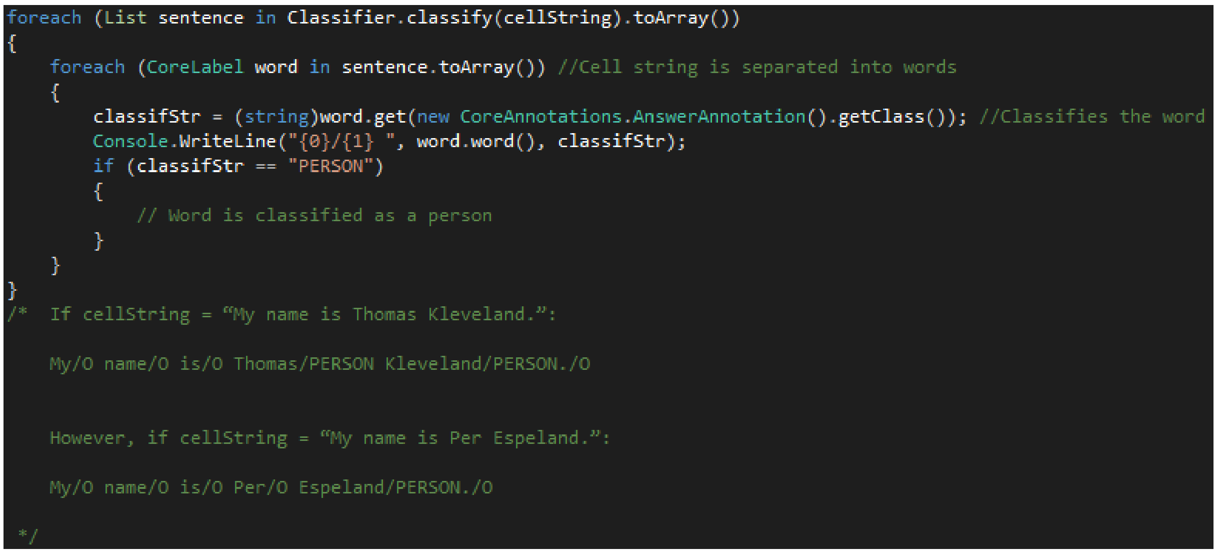
It was however not possible to access the matching weights directly, e.g. "*manager^0.8333333*".

3.2 Stanford NER

The implementation is available in a zipped file from the Stanford NER website [8]. Direct download location: http://nlp.stanford.edu/software/stanford-ner-2015-01-29.zip (150 MB). The java-library and classifiers are inside this file, however it’s mostly the classifiers that takes up space.

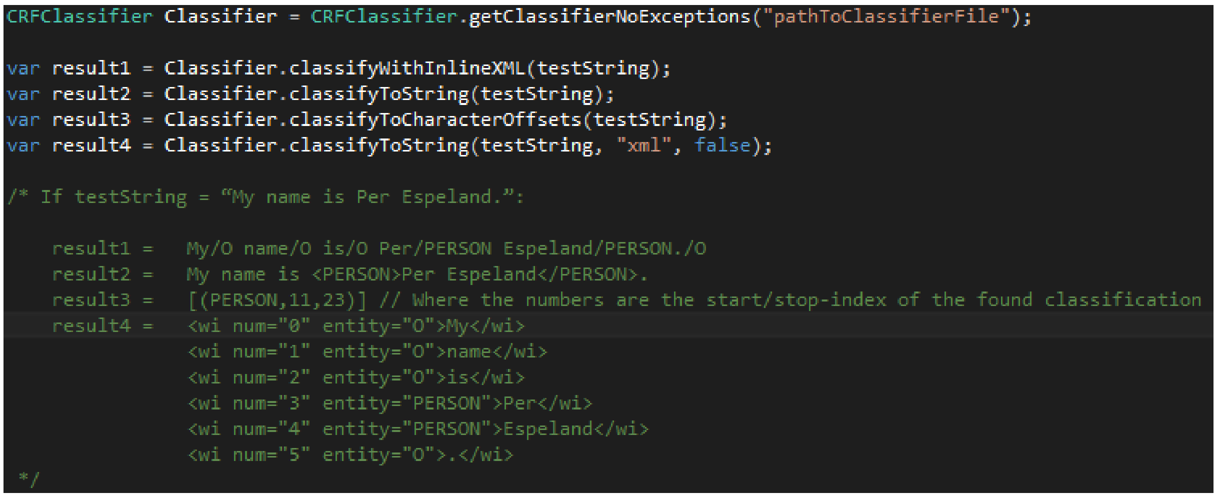
For the person name recognition, only the 3-class model is needed (*english.all.3class.distsim.crf.ser.gz*). For the date and currency recognition, the 7-class model is needed (*english.all.7class.distsim.crf.ser.gz*). The classifier takes some seconds to load into the memory. The implementation was done in C# with the use of a ported java-library [23].

The following examples are inspired by the examples provided on the webpage of the java-library port [23]. Using the implementation, there are different ways of classifying a string. One way is to take the entire string and classify it, the other way is classifying the string word by word. It was needed to extract the classification, and the word by word alternative provided the easiest method of doing so as it iterated through the entities instead of cells. What was discovered when using the word by word classification was that Stanford NER also takes context into account. For example, when running the string “*My name is Per Espeland*” through the both of the ways of classifying a string, the word by word method did not tag *Per* as a person. Classifying the entire string however, correctly tagged *Per* as a person because it had contextual knowledge compared to the word by word classification method.



```
foreach (List sentence in Classifier.classify(cellString).toArray())
{
    foreach (CoreLabel word in sentence.toArray()) //Cell string is separated into words
    {
        classifStr = (string)word.get(new CoreAnnotations.AnswerAnnotation().getClass()); //Classifies the word
        Console.WriteLine("{0}/{1} ", word.word(), classifStr);
        if (classifStr == "PERSON")
        {
            // Word is classified as a person
        }
    }
}
/*  If cellString = “My name is Thomas Kleveland.”:

    My/O name/O is/O Thomas/PERSON Kleveland/PERSON./O


    However, if cellString = “My name is Per Espeland.”:

    My/O name/O is/O Per/O Espeland/PERSON./O

 */
```

Figure 21: Classifying word by word with Stanford NER



```
CRFClassifier Classifier = CRFClassifier.getClassifierNoExceptions("pathToClassifierFile");

var result1 = Classifier.classifyWithInlineXML(testString);
var result2 = Classifier.classifyToString(testString);
var result3 = Classifier.classifyToCharacterOffsets(testString);
var result4 = Classifier.classifyToString(testString, "xml", false);

/* If testString = "My name is Per Espeland.":

    result1 =   My/O name/O is/O Per/PERSON Espeland/PERSON./O
    result2 =   My name is <PERSON>Per Espeland</PERSON>.
    result3 =   [(PERSON,11,23)] // Where the numbers are the start/stop-index of the found classification
    result4 =   <wi num="0" entity="O">My</wi>
                <wi num="1" entity="O">name</wi>
                <wi num="2" entity="O">is</wi>
                <wi num="3" entity="PERSON">Per</wi>
                <wi num="4" entity="PERSON">Espeland</wi>
                <wi num="5" entity="O">.</wi>
 */
```

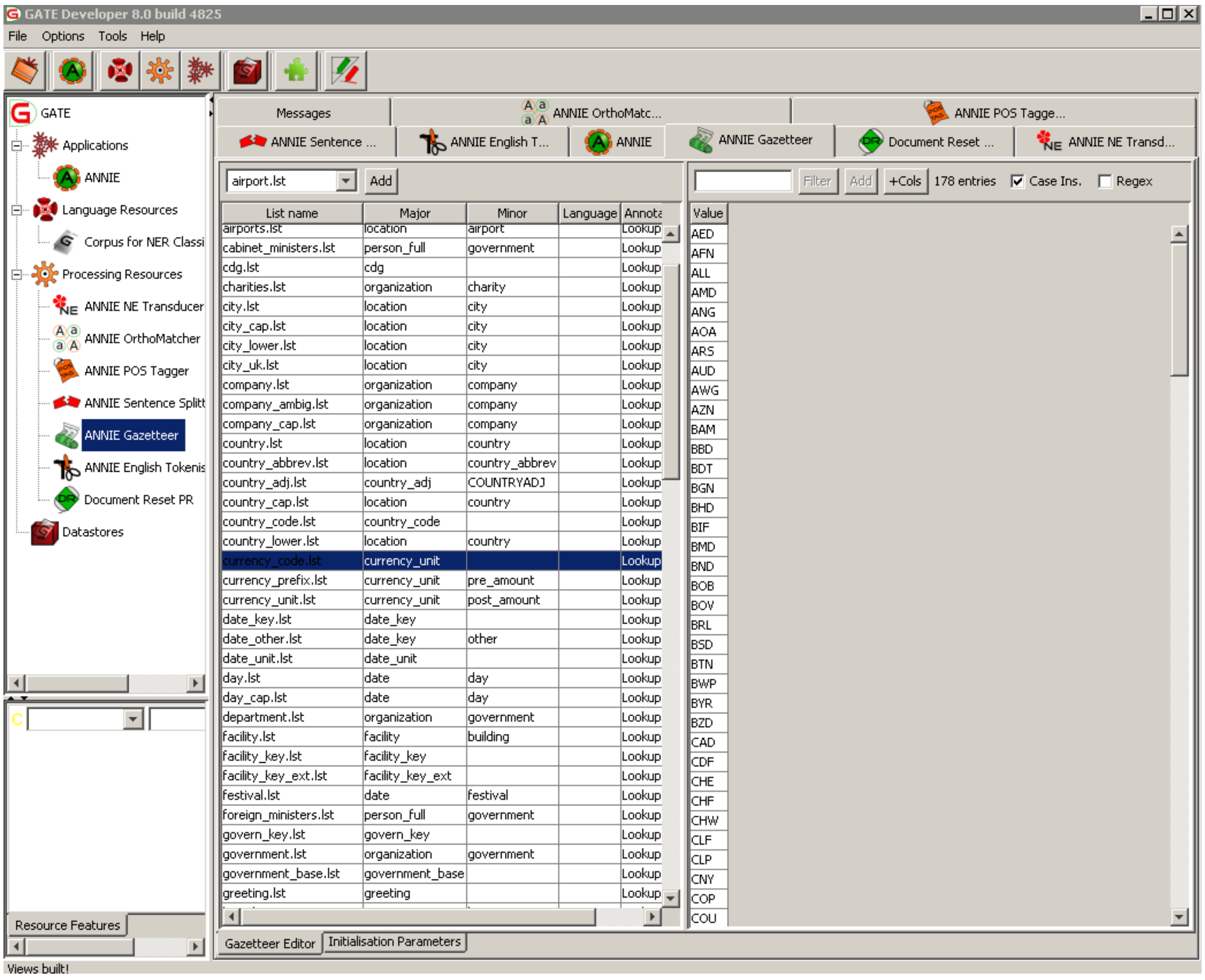
Figure 22: Classifying a string with Stanford NER

Where *O* (means *Other*) and *PERSON* are the classifications. The example of the result mentioned in chapter 2.2.3 was created using the word by word method as the knowledge of the difference between the methods was found at a later point in time.

For using the NER in java, the official java-demo-file can be found at http://nlp.stanford.edu/software/ner-example/NERDemo.java.

3.3 ANNIE with GATE Developer

This chapter assumes knowledge of information presented in chapter 2.4.2. The file, *number.jape*, was used as a starting point when creating new *JAPE*-rules in the file *currency.jape*. For the file to be discovered in the *ANNIE NE Transducer*, the filename was needed to be listed in the *main.jape*-file. A gazetteer list of the currency codes [16] was added to *ANNIE* under the *Major*-type *currency_unit*.



airport.lst

List name	Major	Minor	Language	Annota
airports.lst	location	airport		Lookup
cabinet_ministers.lst	person_full	government		Lookup
cdg.lst	cdg			Lookup
charities.lst	organization	charity		Lookup
city.lst	location	city		Lookup
city_cap.lst	location	city		Lookup
city_lower.lst	location	city		Lookup
city_uk.lst	location	city		Lookup
company.lst	organization	company		Lookup
company_ambig.lst	organization	company		Lookup
company_cap.lst	organization	company		Lookup
country.lst	location	country		Lookup
country_abbrev.lst	location	country_abbrev		Lookup
country_adj.lst	country_adj	COUNTRYADJ		Lookup
country_cap.lst	location	country		Lookup
country_code.lst	country_code			Lookup
country_lower.lst	location	country		Lookup
currency_code.lst	currency_unit			Lookup
currency_prefix.lst	currency_unit	pre_amount		Lookup
currency_unit.lst	currency_unit	post_amount		Lookup
date_key.lst	date_key			Lookup
date_other.lst	date_key	other		Lookup
date_unit.lst	date_unit			Lookup
day.lst	date	day		Lookup
day_cap.lst	date	day		Lookup
department.lst	organization	government		Lookup
facility.lst	facility	building		Lookup
facility_key.lst	facility_key			Lookup
facility_key_ext.lst	facility_key_ext			Lookup
festival.lst	date	festival		Lookup
foreign_ministers.lst	person_full	government		Lookup
govern_key.lst	govern_key			Lookup
government.lst	organization	government		Lookup
government_base.lst	government_base			Lookup
greeting.lst	greeting			Lookup

178 entries

Value
AED
AFN
ALL
AMD
ANG
AOA
ARS
AUD
AWG
AZN
BAM
BBD
BDT
BGN
BHD
BIF
BMD
BND
BOB
BOV
BRL
BSD
BTN
BWP
BYR
BZD
CAD
CDF
CHE
CHF
CHW
CLF
CLP
CNY
COP
COU

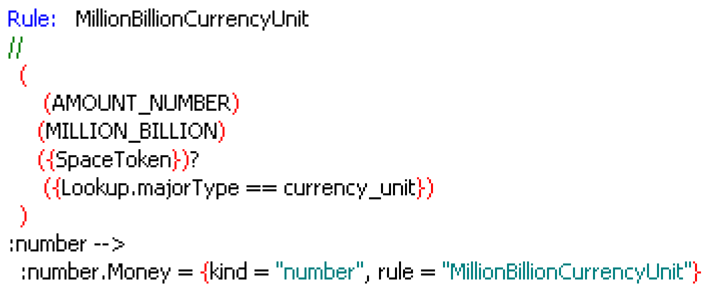
Figure 23: Currency code list added to ANNIE Gazetteer

An example of a rule to match a token with one of the gazetteer lists in the *Major*-type *curreny_unit*:

```
Rule:  CurrencyUnit
//
 (
   ({Lookup.majorType == currency_unit})
 )
:number -->
 :number.Money = {kind = "number", rule = "CurrencyUnit"}
```

Figure 24: Currency unit gazetteer lookup

This successfully discovered all tokens that matched any entries in the gazetteer lists in the *Major*-type *currency_unit*. However, when the attempt was made to add a prefix to the lookup so that it would match *MNOK* or *MMNOK* instead of just *NOK*, it did not have the expected result. The reason for this was the close dependency of tokens in the programming language. Only individual tokens could be compared to gazetteer lists, and as *MNOK* or *MMNOK* were individual tokens, they would not get matched as there were no entries in the gazetteer lists equal to that string. The rule that failed to successfully match a currency notation:



```
Rule:  MillionBillionCurrencyUnit
//
 (
    (AMOUNT_NUMBER)
   (MILLION_BILLION)
   ({SpaceToken})?
    ({Lookup.majorType == currency_unit})
 )
:number -->
 :number.Money = {kind = "number", rule = "MillionBillionCurrencyUnit"}
```

Figure 25: MillionBillionCurrencyUnit rule

Where *AMOUNT_NUMBER* (default macro from *number.jape*) was the number, *MILLION_BILLION* (default macro from *number.jape* with a few more additions) was the amount multiplicator (e.g. *M, MM, K, etc.*), the optional *SpaceToken* represents a space and the lookup is the gazetteer lists with the *Major*-type *currency_unit*. The rule would successfully match “2800 M NOK” or “2800M NOK”, but not “2800 MNOK” or “2800MNOK”. The reason being the problem mentioned earlier in this chapter and chapter 2.4.2. An alternative solution to this could be to add all of the currency codes with all of the different amount multiplicators. This list would however get very large (the count of currency codes times the count of amount multiplicators). The reason for trying out *ANNIE* was because it looked like it was possible to create a rule using a combination of some kind of regex and gazetteer list. The difference was that a regex works on a simple word (e.g. *(M)?NOK* would cover both *MNOK* and *NOK*), but the Jape-language had limitations when trying to combine strings before comparing it to a gazetteer list.

3.4 OYSTER

The OYSTER demonstration runs, documentation and source code was downloaded from http://sourceforge.net/projects/oysterer/files/, and then navigating to “OYSTER_3.3/Download All/ OYSTER_v3.3_Download_All..zip”.

The source of OYSTER, including all of the files created/edited in the implementation comes with the thesis submission.

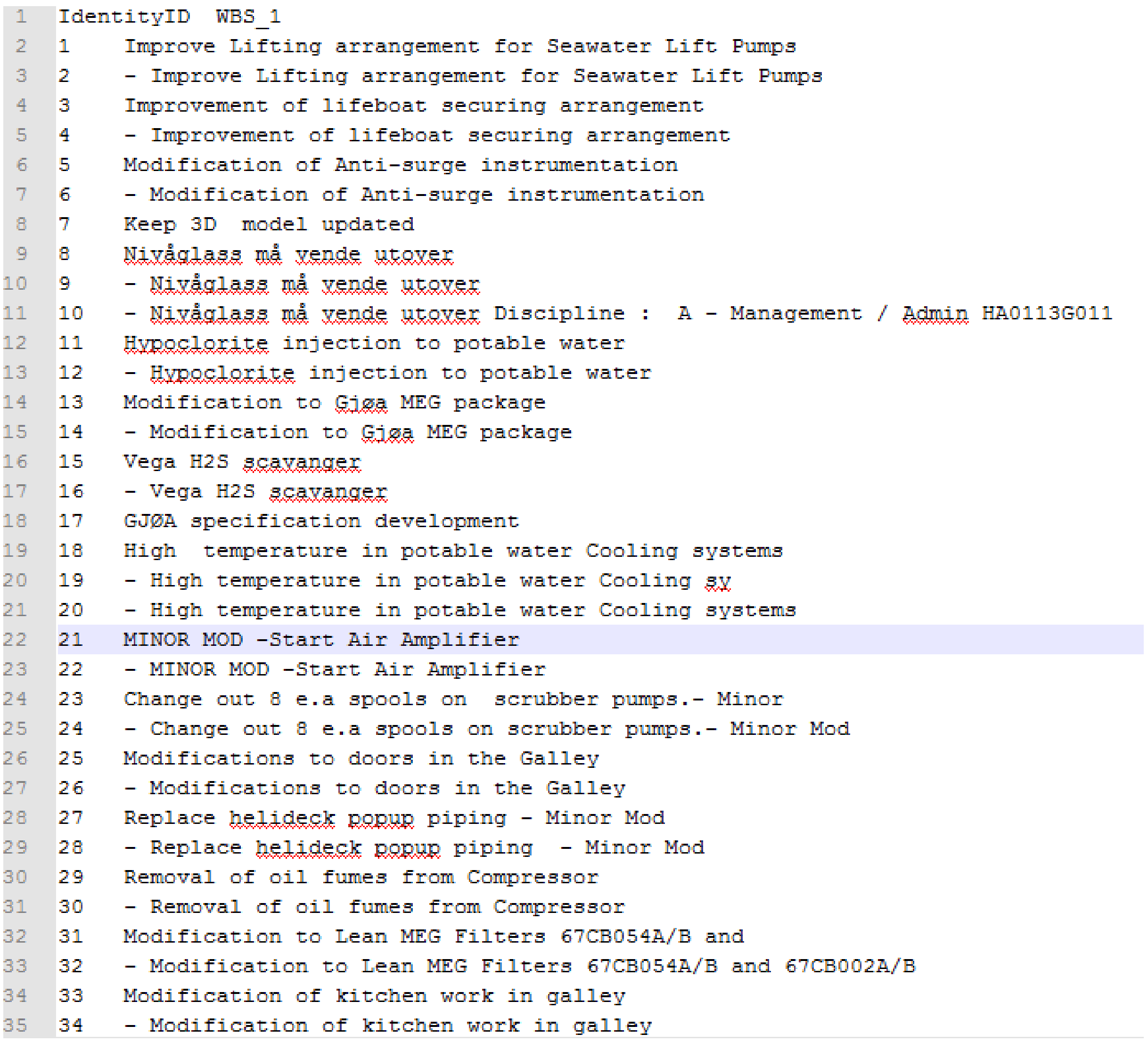
3.4.1 Basic knowledge

The OYSTER implementation comes with different run-modes [24]:

Run-Mode	**Short description**
Merge-Purge	Entity references are systematically compared to each other and separated into clusters of equivalent records.
Identity Capture	System builds a set of identities from the references it processes rather than starting with a known set of identities.
Reference to Reference Assertion	A process of forcing references to match even when no defined match rules would be able to bring them together. The forced matches are based off of previous user knowledge of the references.
Identity Resolution	Only matches the references to the input identities, not the references itself. The input identities are specified in an .idty-file.

Identity Update	Both takes input identities as input and updates these identities with any new information presented as an updated .idty-file.
Reference to Structure Assertion	Forces references to be matched with an existing identity in the .idty-file by using the OysterID
Structure to Structure Assertion	Used to fix false negative matches produced in previous runs by force re-match references.
Structure Split Assertion	Used to fix false positive matches and does the opposite of what the "Structure to Structure Assertion" does. It forces an identity structure in an existing .idty-file to divide into two or more identity structures.

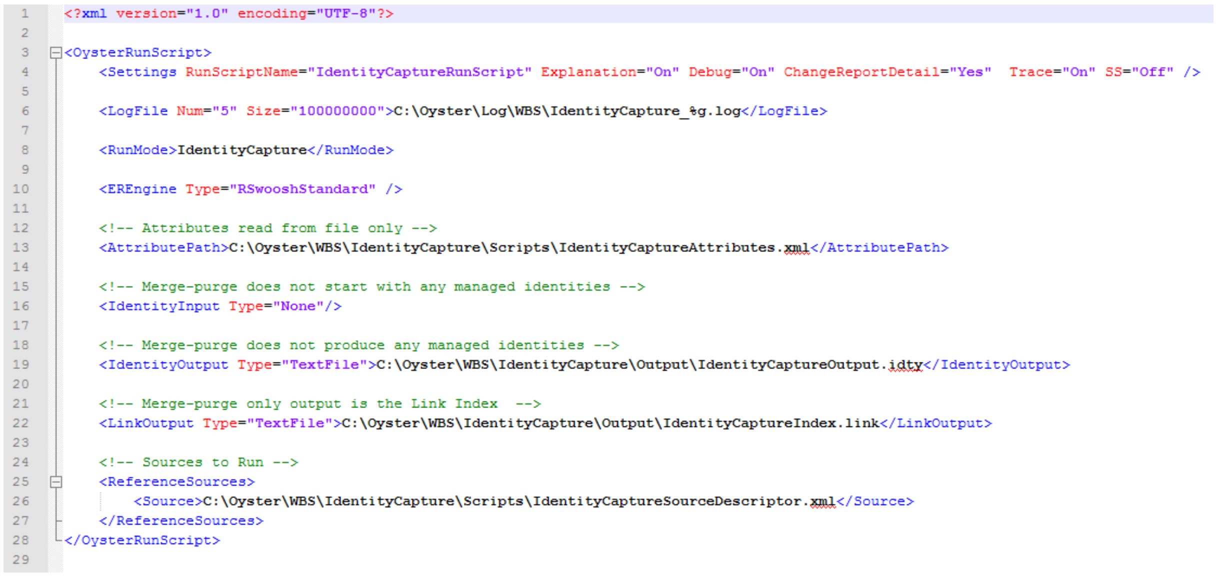
These run-modes were presented as demonstration runs (Run001-008). The run-mode, Identity Capture, was tested using a spreadsheet of WBS project titles converted to a text file with a tabular delimiter (*WBS (unique, without WBSID).txt*):



```
IdentityID  WBS_1
1    Improve Lifting arrangement for Seawater Lift Pumps
2    - Improve Lifting arrangement for Seawater Lift Pumps
3    Improvement of lifeboat securing arrangement
4    - Improvement of lifeboat securing arrangement
5    Modification of Anti-surge instrumentation
6    - Modification of Anti-surge instrumentation
7    Keep 3D  model updated
8    Nivåglass må vende utover
9    - Nivåglass må vende utover
10   - Nivåglass må vende utover Discipline :  A - Management / Admin HA0113G011
11   Hypoclorite injection to potable water
12   - Hypoclorite injection to potable water
13   Modification to Gjøa MEG package
14   - Modification to Gjøa MEG package
15   Vega H2S scavanger
16   - Vega H2S scavanger
17   GJØA specification development
18   High  temperature in potable water Cooling systems
19   - High temperature in potable water Cooling sy
20   - High temperature in potable water Cooling systems
21   MINOR MOD -Start Air Amplifier
22   - MINOR MOD -Start Air Amplifier
23   Change out 8 e.a spools on  scrubber pumps.- Minor
24   - Change out 8 e.a spools on scrubber pumps.- Minor Mod
25   Modifications to doors in the Galley
26   - Modifications to doors in the Galley
27   Replace helideck popup piping - Minor Mod
28   - Replace helideck popup piping  - Minor Mod
29   Removal of oil fumes from Compressor
30   - Removal of oil fumes from Compressor
31   Modification to Lean MEG Filters 67CB054A/B and
32   - Modification to Lean MEG Filters 67CB054A/B and 67CB002A/B
33   Modification of kitchen work in galley
34   - Modification of kitchen work in galley
```

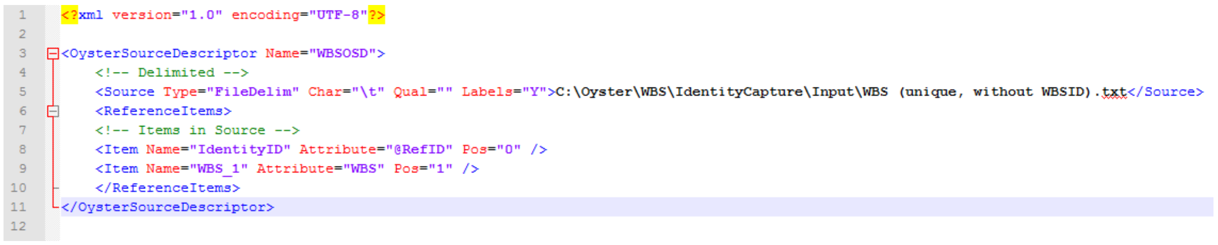
Figure 26: A snapshot of the Identity Capture test data

A run script that references the *"Source Descriptor"*- and *"Attribute*"-files (mentioned later in this chapter) was created. It contains the locations of the output files and the log-file, as well as the run-mode. It is also possible to specify the Entity Resolution (ER) engine used to process the references. The default is the R-Swoosh algorithm:



```xml
<?xml version="1.0" encoding="UTF-8"?>

<OysterRunScript>
    <Settings RunScriptName="IdentityCaptureRunScript" Explanation="On" Debug="On" ChangeReportDetail="Yes"  Trace="On" SS="Off" />

    <LogFile Num="5" Size="100000000">C:\Oyster\Log\WBS\IdentityCapture_%g.log</LogFile>

    <RunMode>IdentityCapture</RunMode>

    <EREngine Type="RSwooshStandard" />

    <!-- Attributes read from file only -->
    <AttributePath>C:\Oyster\WBS\IdentityCapture\Scripts\IdentityCaptureAttributes.xml</AttributePath>

    <!-- Merge-purge does not start with any managed identities -->
    <IdentityInput Type="None"/>

    <!-- Merge-purge does not produce any managed identities -->
    <IdentityOutput Type="TextFile">C:\Oyster\WBS\IdentityCapture\Output\IdentityCaptureOutput.idty</IdentityOutput>

    <!-- Merge-purge only output is the Link Index  -->
    <LinkOutput Type="TextFile">C:\Oyster\WBS\IdentityCapture\Output\IdentityCaptureIndex.link</LinkOutput>

    <!-- Sources to Run -->
    <ReferenceSources>
        <Source>C:\Oyster\WBS\IdentityCapture\Scripts\IdentityCaptureSourceDescriptor.xml</Source>
    </ReferenceSources>
</OysterRunScript>
```

A source descriptor .xml-file referencing the input references and the items (columns in this case) in the input was created, called " *IdentityCaptureSourceDescriptor.xml*”:

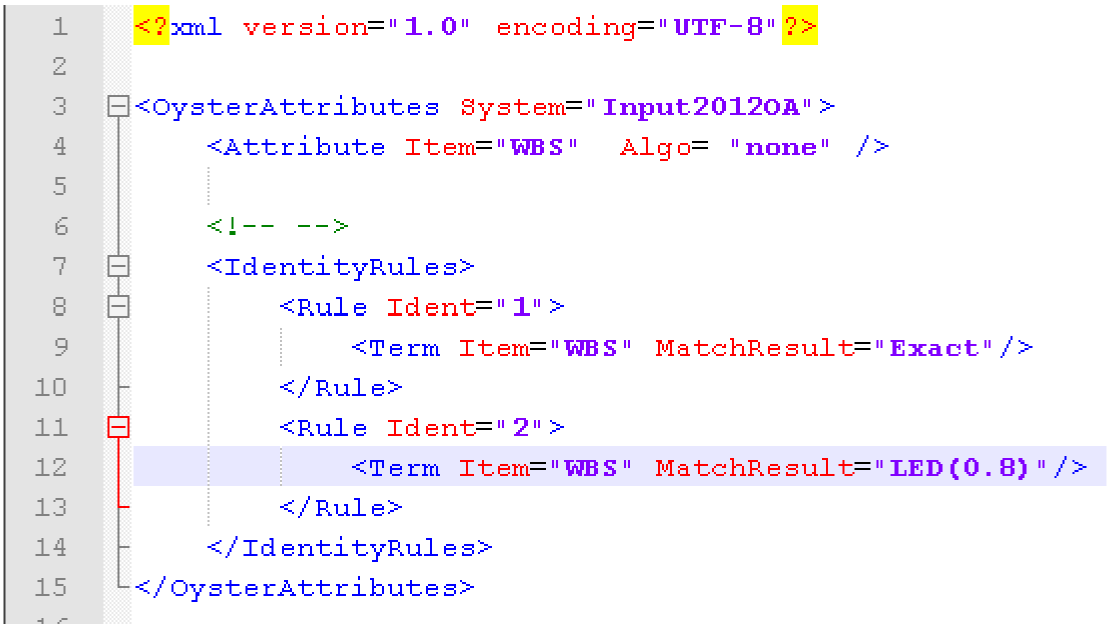


```xml
<?xml version="1.0" encoding="UTF-8"?>

<OysterSourceDescriptor Name="WBSOSD">
    <!-- Delimited -->
    <Source Type="FileDelim" Char="\t" Qual="" Labels="Y">C:\Oyster\WBS\IdentityCapture\Input\WBS (unique, without WBSID).txt</Source>
    <ReferenceItems>
    <!-- Items in Source -->
    <Item Name="IdentityID" Attribute="@RefID" Pos="0" />
    <Item Name="WBS_1" Attribute="WBS" Pos="1" />
    </ReferenceItems>
</OysterSourceDescriptor>
```

Figure 27: Identity Capture OYSTER Source Descriptor

The absolute path for the input-file and tabular delimiter is specified. The column-names are linked to the attributes in the “*Attributes*”-file mentioned later in this chapter.

An example of the “*Attributes*”-file when fuzzy matching with the use of *Levenshtein Edit Distance*:



```xml
<?xml version="1.0" encoding="UTF-8"?>

<OysterAttributes System="Input2012OA">
    <Attribute Item="WBS"  Algo= "none" />

    <!-- -->
    <IdentityRules>
        <Rule Ident="1">
            <Term Item="WBS" MatchResult="Exact"/>
        </Rule>
        <Rule Ident="2">
            <Term Item="WBS" MatchResult="LED(0.8)"/>
        </Rule>
    </IdentityRules>
</OysterAttributes>
```

Figure 28: Identity Capture OYSTER attributes using LED

Using this file, OYSTER will match an entity when it is exactly the same and when the normalized *Levenshtein Edit Distance* threshold is at least *0.8* (the threshold has to be between *0* and *1*, where *1* is an exact match).

3.4.2 Custom matching algorithms

As mentioned in the reference guide: *"OYSTERs default matching algorithm supports the above matching codes but this can be extended by the user by extending the base class OysterComparator.java as a new class with a name starting with "OysterCompare" and implementing the method String: getMatchCode(String, String)."* [21].

To be able to align the entity matching with the rules mentioned in chapter 2.4, two custom matching algorithms were implemented. Using the source code for OYSTER, the "*OysterCompareDefault.java*" was copied and renamed "*OysterCompareCustom.java*", where this file was edited to include the new matching algorithms.

3.4.2.1 CUSTOMLED (*ledThreshold*, *charType*, *useRestAlphaExact*)

The parameters:

Parameter	**Description**
ledThreshold	The Levenshtein Edit Distance threshold. Must be between 0 and 1, where 1 is an exact match. The same as for the default Levenshtein Edit Distance algorithm.
charType	Instead of using the Levenshtein Edit Distance on the entire string, the algorithm uses it on parts of the string instead. Which part is defined by the "*charType*"-attribute and can either be "*LETTER*" or "*DIGIT*". Example: If *charType* = "*LETTER*" and the testing string is "*Management and Admin 2015*", the algorithm would do a Levenshtein Edit Distance test on "*ManagementandAdmin*" vs the letters of the entity being tested against and not on the digits.

useRestAlphaExact	A boolean determining whether to do an exact match on the remaining alphanumeric-characters (digits or letters) that did not get fuzzy matched. E.g.: The remaining characters of "*Management and Admin 2015*" where *charType* = "*LETTER*" would be "*2015*" and where *charType* = "*DIGIT*" would be "*ManagementandAdmin*".

Using this algorithm it was possible to correctly match project titles that had spelling errors, but still exclude project titles with a different year.

3.4.2.2 LEFTCONTAINS (*useOnlyAlpha, minLength*)

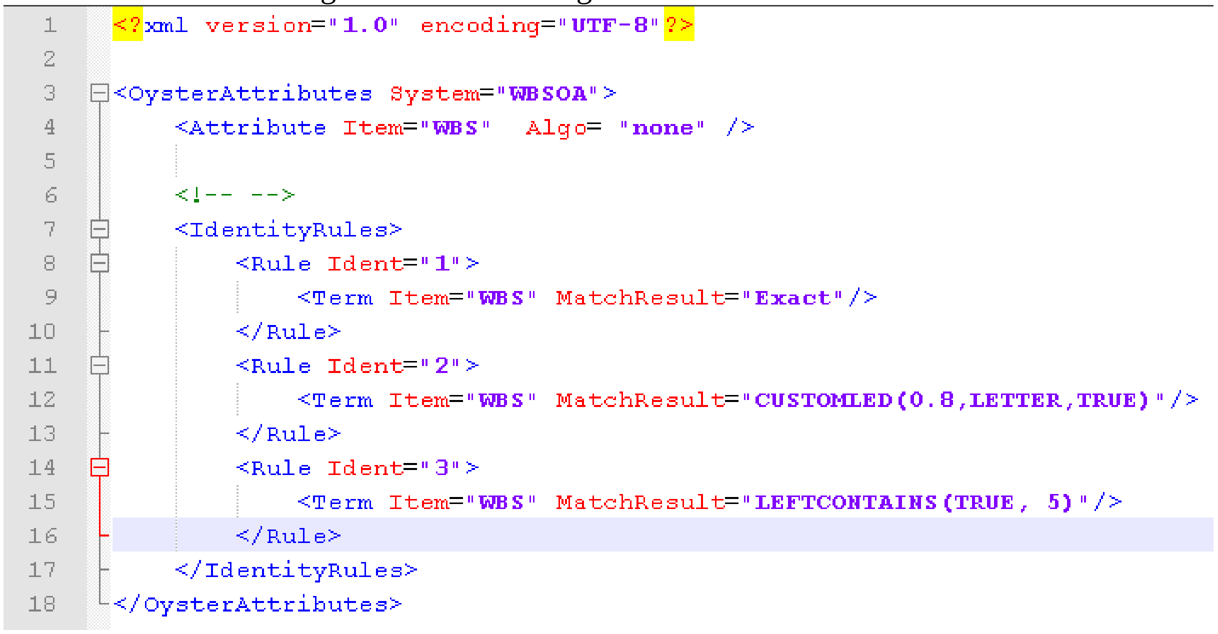
The algorithm checks if the comparing strings are contained in the start of one of them. E.g.: "*TEST*" would result in a match when compared to "*TESTING*", but "*TEST*" would not result in a match when compared to "*PTESTING*".

The parameters:

Parameter	**Description**
useOnlyAlpha	A boolean determining whether to only use the alphanumeric characters of the comparing strings. This will exclude most symbols.
minLength	The minimum length required for the comparing strings. This is to be able to prevent strings like "*M*" causing all strings that starts with "*M*" to be matched as the same entity.

Using this algorithm it was possible to correctly match project titles that had "cut-offs".

The "Attribute"-file using the new custom algorithms:



```
<?xml version="1.0" encoding="UTF-8"?>

<OysterAttributes System="WBSOA">
    <Attribute Item="WBS"  Algo= "none" />

    <!-- -->
    <IdentityRules>
        <Rule Ident="1">
            <Term Item="WBS" MatchResult="Exact"/>
        </Rule>
        <Rule Ident="2">
            <Term Item="WBS" MatchResult="CUSTOMLED(0.8,LETTER,TRUE)"/>
        </Rule>
        <Rule Ident="3">
            <Term Item="WBS" MatchResult="LEFTCONTAINS(TRUE, 5)"/>
        </Rule>
    </IdentityRules>
</OysterAttributes>
```

References

[1] Wikipedia, “Approximate string matching - Wikipedia, the free encyclopedia,” 15 April 2015. [Online]. Available: https://en.wikipedia.org/wiki/Approximate_string_matching. [Accessed 15 June 2015].

[2] Wikipedia, “Regular expression - Wikipedia, the free encyclopedia,” Wikipedia, the free encyclopedia, 05 June 2015. [Online]. Available: http://en.wikipedia.org/wiki/Regular_expression. [Accessed 11 June 2015].

[3] G. Skinner and gskinner team, “RegExr: Learn, Build, & Test RegEx,” [Online]. Available: http://www.regexr.com/.

[4] S. Toarca, “Debuggex: Online visual regex tester. JavaScript, Python, and PCRE.,” Debuggex.com, [Online]. Available: https://www.debuggex.com/.

[5] Wikipedia, "Levenshtein distance - Wikipedia, the free encyclopedia," 21 April 2015. [Online]. Available: http://en.wikipedia.org/wiki/Levenshtein_distance. [Accessed 10 June 2015].

[6] Wikipedia, “Lucene - Wikipedia, the free encyclopedia,” Wikipedia, the free encyclopedia, 09 June 2015. [Online]. Available: http://en.wikipedia.org/wiki/Lucene. [Accessed 10 June 2015].

[7] Apache Software Foundation, “FuzzyQuery (Lucene 4.3.0 API),” [Online]. Available: https://lucene.apache.org/core/4_3_0/core/org/apache/lucene/search/FuzzyQuery.html.

[8] The Stanford Natural Language Processing Group, “The Stanford NLP (Natural Language Processing) Group,” [Online]. Available: http://nlp.stanford.edu/software/CRF-NER.shtml. [Accessed 10 June 2015].

[9] Gliffy, “Online Diagram Software and Flow Chart Software - Gliffy,” Gliffy, 2015. [Online]. Available: https://www.gliffy.com.

[10] Wikipedia, “General Architecture for Text Engineering - Wikipedia, the free encyclopedia,” 12 June 2015. [Online]. Available: https://en.wikipedia.org/wiki/General_Architecture_for_Text_Engineering. [Accessed 14 June 2015].

[11] GATE, “GATE.ac.uk - sale/tao/splitch6.html,” [Online]. Available: https://gate.ac.uk/sale/tao/splitch6.html#chap:annie.

[12] D. Thakker, T. Osman and P. Lakin, “GATE JAPE Grammar Tutorial,” 27 February 2009. [Online]. Available: https://gate.ac.uk/sale/thakker-jape-tutorial/GATE%20JAPE%20manual.pdf.

[13] H. Cunningham, D. Maynard and K. Bontcheva, Text Processing with GATE (Version 6), GATE, 2011.

[14] H. Cunningham, V. Tablan, A. Roberts and K. Bontcheva, “PLOS Computational Biology: Getting More Out of Biomedical Documents with GATE's Full Lifecycle Open Source Text Analytics,” 2013. [Online]. Available: http://tinyurl.com/gate-life-sci/. [Accessed 2015].

[15] L. Tupone, “What is "The Best" U.S. Currency RegEx? - Stack Overflow,” 15 December 2014. [Online]. Available: http://stackoverflow.com/questions/354044/what-is-the-best-u-s-currency-regex#answer-27491430. [Accessed 15 June 2015].

[16] ISO 4217 maintenance agency (MA), SIX Interbank Clearing, “Current currency & funds code list - ISO Currency,” 2015. [Online]. Available: http://www.currency-iso.org/en/home/tables/table-a1.html. [Accessed April 2015].

[17] Wikipedia, “Currency symbol - Wikipedia, the free encyclopedia,” 14 June 2015. [Online]. Available: https://en.wikipedia.org/wiki/Currency_symbol. [Accessed 15 June 2015].

[18] J. R. Talburt, in *Entity Resolution and Information Quality,* Morgan Kaufmann; 1 edition (December 22, 2010), 2010, pp. 1, 63-99, 157-172.

[19] F. Kobayashi and J. Talburt, Introduction to Entity Resolution with OYSTER v3.3, University of Arkansas at Little Rock, 2012.

[20] Wikipedia, “Record linkage - Wikipedia, the free encyclopedia,” 4 June 2015. [Online]. Available: https://en.wikipedia.org/wiki/Record_linkage. [Accessed 15 June 2015].

[21] F. Kobayashi and J. Talburt, in *OYSTER v3.3 Reference Guide,* University of Arkansas at Little Rock, 2013, pp. 22-26.

[22] F. Sabar, “Fazlan's Blog Spot: Apache Lucene Tutorial: Indexing Microsoft Documents,” 7 June 2012. [Online]. Available: http://fazlansabar.blogspot.no/2012/06/apache-lucene-tutorial-indexing.html.

[23] The Stanford Natural Language Processing Group, “Stanford Named Entity Recognizer (NER) for .NET,” [Online]. Available: http://sergey-tihon.github.io/Stanford.NLP.NET/StanfordNER.html.

[24] F. Kobayashi, OYSTER v3.3 Demonstration Runs User Guide, University of Arkansas at Little Rock, 2013.

A Zip-file content

The zip file contains the implementations done including some of the gazetteer lists. As I did not have time to format the C# implementations, they do not look very pretty.