Data Extraction

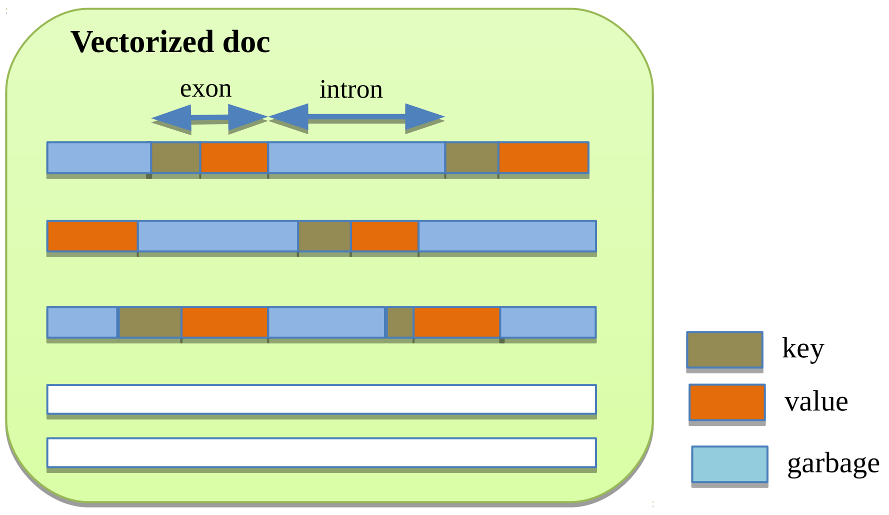
Bruno Apolloni, Simone Bassis and Rocco de Rosa

Table of Contents

1 Basic idea

The goal is to split a vectorized document in a sequence of meaningful segments and garbage, where the former in turn are split in keys, i.e. entries of our interest, and values representing their instantiation in the document.

Drawing analogy from genomic sequences, we may consider meaningful sequences the analogous of exons and garbage the analogous of introns, getting the visual representation of Figure 1.



Example

Your Contract Ref: pami of new mexico .Our client declares Participation: 19.50% of 100% The accident has been registered with Our Reference:Aon Ben with the following date of loss: 12/07/2010

Figure 1. *The value extraction problem.*

Its achievement entails a correct identification of exons in their parts:

- keys from among a set of synonyms of different lengths, and
- values as limited suffixes of keys

As for the keys, the idea is to search for synonyms, where each synonym (either a singleton or group of words) generates a possible segment within the text.

By scrolling the text, all potential solutions (those that are represented by the synonyms of the keys whose words are “similar” to the word currently read) are “hooked”. At each step new solutions (possible correct matches) are activated, whereas invalid solutions are deleted according to suited criteria. Once a solution has been “hooked”, its value is represented by the sequence of words we read after the solution until a stopping criterion is reached (e.g. a new solution is “hooked” or a punctuation mark reached, etc.).

1.1 Glossary

Term	Description
key	the field of the database whose value we are looking for. In particular, the keys considered in this release belong to the set {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}.
synonym	a string (or a set of strings) expressing a key in the document.
atom	each of the words constituting a synonym
value	a set of words associated at a key (or one of its synonym) in the document
document	text to extract data, i.e. couples key-value
session	abstraction of a document practice. A document may be constituted by several practices: to recognize when a new practice takes place in the document, every time we discover a key which has already been processed in the same document, we open a new session.
text word	Each of the words constituting the document
solution	A portion of document matching a given synonym together with the corresponding identified value

1.2 Main Parts

The problem is decomposed into two main parts:

Part I: we discover all values recognized as a set of words matching a synonym's key. If we find a second value for the same key, a new session will be opened.

Part II: the key values found in the first step may be incorrect or spurious; hence they require a cleaning and filtering process.

Each of the steps mentioned above are further divided into sub-phases.

2 PART I

Before stating the implementation and development issues related to the in-document key-value discovering, we will get a picture of the whole procedure, with the aim of highlighting the main steps to be followed in order to achieve our goal. In turn, these steps need some care in order to be fully appreciated and understood: this is why we will maintain this short description on a high level, providing in the meanwhile some examples and hints that prove essential for guaranteeing comprehensibility of the next sections.

2.1 General description of the whole procedure

Let's start with a leading example. Consider the simple document[1] reported in Example I.1, where for the sake of simplicity we omit introns:

DOC: "your contract ref: pami of new mexico . participation: 19.50% of 100% our reference: aon ben date of loss: 12/07/2010"

***Example I.1**. Document used as leading example.*

Our aim is to retrieve within the document those keys belonging to a list provided in input to the user (see Table I.1) where to each key (e.g. Key1, Key2, …) a list of synonyms is associated describing the possible ways wherein the former can occur within the document.

Keys	Synonyms			
Key1	your contract	your contract ref	your reference	
Key2	our contract	our contract ref	our reference	
Key3	contract	contract name	tty name	treaty
Key4	your share	signed line	participation	participant's
Key5	date of loss	loss date	date of claim	

***Table I.1.** Original keys with associated synonyms.*

For instance, Key1 may appear in the text either through the synonym "your contract ref" or "your reference".

To retain a low computational burden, considering the huge number of documents to be processed, we developed a one-pass method, i.e. an algorithm able to find all the sought occurrences in a single scan. Reading the text word-by-word, it should be able to recognize both the keys provided in input (for instance those listed in Table I.1) together with their corresponding value (typically the set of words following the discovered key).

[1] As explained in a following remark it is warmly recommended to convert to lowercase all the strings encountered during the data extraction. Hence, from here on, all strings contained in examples and tables will be reported in lowercase.

A reasonable way to proceed is that of considering the text to be processed as a sequence of *words*. From the leading example we will split the document in Example I.1 as follows (details on the splitting delimiters will be provided in a later section):

t = 1: your, *t* = 2: contract, *t* = 3: ref:, ...

where the counter *t* reads as a timestamp variable for the document scan. Analogously, to maintain consistency in the search procedure, all the synonyms should be considered as a set words (*atoms*), so that Table I.1 reads now as follows:

Key1	your	contract	your	contract	ref	your	reference
Key2	our	contract	our	contract	ref	our	reference
Key3	contract	contract	name	tty	name	treaty	
Key4	your	share	signed	line	participation	participant’s	
Key5	date	of	loss	loss	date	date	of claim

***Table I.2**. Synonyms split in atoms together with the key they are related to.*

We are now allowed (at each step *t* = 1, 2, …) to compute the similarity between the current word at time *t* and each atoms of Table I.2. The idea is to single out all atoms most similar to the current word (for instance having distance less than a given threshold). In turn, the keys associated to the synonyms whose atoms have been selected will be candidate to become possible solutions, i.e. keys we are asked to associate the corresponding values. Namely, these synonyms, called *candidate synonyms*, will be stored in a data structure (*stack of solution*) whose content will dynamically change with the document scan.

In fact, not all candidate synonyms correspond to valid solutions: we must check that all atoms of a candidate synonym are encountered sequentially in the text, respecting the occurrence order of the atoms. For instance, to validate the synonym “your contract ref”, we should initially find in the text the word “your”, followed by “contract” and finally by “ref”. But if we were to find initially the word “contract”, this partial solution should be discarded. The reader may imagine the huge variety of controls that a candidate key must pass in order to become valid. Think for instance to synonyms which are substrings of other synonyms: if we were to find in the text the following sequence of words: “... your contract ref: pami of new mexico ...”, after the reading of “your contract” both synonyms “your contract” and “your contract ref” should be candidate to become valid solution; however, after having read the word “ref”, the former will be discarded. Moreover, note the subtle fact that synonyms which may be composed in other synonyms may correspond even to different keys. We will call *valid solutions* those candidate solutions which pass the aforementioned controls. In other words, the *stack of solutions* will be used to dynamically update the list of all candidate solutions: when a entry will pass all the expected controls, it will be marked as valid solution.

Once a valid solution has been discovered, i.e. all atoms have been recognized in the right order within the document, we start the identification of the corresponding value, which we assume to be composed by all those words immediately following the valid synonym. In detail, a valid solution become an *active solution* exactly in the next timestamp, until its value has been completely discovered, i.e. a suitable stop criterion has been reached. This may read as the activation of another synonym, or the presence of a punctuation mark.

The final result we expect from Example I.1 having in input keys and synonyms of Table I.1 is reported in the following report:

Key 1	pami of new mexico
Key 4	19.50% of 100%
Key 2	aon ben
Key 5	12/07/2010

Table I.3. *Expected final report.*

We are now ready to depict the data structures needed in both the *initialization phase* (depending uniquely on the key-synonyms table, as Table I.1 is) and the *processing phase* (wherein the key-value pairs must be discovered in a document provided in input).

Remark. All the descriptive steps of the proposed methodology will be equipped with a set of example files (*check points*) useful to implement and validate the algorithm.

2.2 Phase 1: initialization

2.2.1 Description

In this phase we will depict all the data structures needed to efficiently compute the distance between the text words and the atoms, and those allowing us to access all the information on the composition of synonyms gradually selected as candidate solutions.

2.2.2 Implementation

Let's start from Tables I.1 and I.2, the former containing all the keys and synonyms given in input to the procedure, the latter splitting each synonym in its constituent atoms, where the split uses the space character as word delimiter.

Remark 1. It is warmly recommended to convert all text and synonyms in lowercase, making the distance computed between words more reliable.

Remark 2. Special care must be devoted to synonyms typed without spaces such as “YourContract” or “ContractName”. Even if the implementation we provide does not care about this issue, a possible way to tackle it would be to add such occurrences to the synonym list of Table I.1.

While scanning the text, some of the synonyms listed in Table I.1 may rise to the rank of candidate solution. Therefore, we must assign to each of them an identification code, storing all the information useful for the next phase where candidate solutions should be validated. In detail:

- we assign at each key in Table I.1 a code (*key_id*) corresponding to the ordinal number of the row where the key lies (e.g. key1→1, key2→2, …);
- we couple each synonym in Table I.1 with a code (*syn_id*) which corresponds to the sequential number of the cell where it lies (e.g. “your contract”→1, “your contract ref”→2, “your reference”→3…).

Moreover, for each synonym we associate to its *syn_id* the *key_id* of the corresponding key together with the number *syn_length* of atoms constituting the synonym (useful to understand when a solution becomes valid and therefore in *activated* state), obtaining the following Table I.4.

syn_id	**key_id**	**syn_length**	
1	1	2	“your contract” (code 1) corresponding to key1 (code 1)
2	1	3	
3	1	2	“your reference” (code 3) related to key1 (code 1)
4	2	2	
5	2	3	
6	2	2	
7	3	1	
8	3	2	
9	3	2	
10	3	1	
11	4	2	
12	4	2	
13	4	1	
14	4	1	
15	5	3	
16	5	2	
17	5	3	

Table I.4. *Synonym information table.*

As described in the previous section, at each step *t* = 1, 2, …, the distance between the current text word and the atoms of Table I.2 are computed. As explained in the next section, such distances will be store in a table having the same number of rows and columns of Table I.2. As each cell contains the distance of the current text work from the atom the cell refers to, in order to go back quickly to the synonyms information table (see Table I.4) useful for both selecting the candidate solution and checking their validity through the implemented reliability controls, the following data structure proves to be very fruitful:

Key1	1	1	2	2	2	3	3
Key2	4	4	5	5	5	6	6
Key3	7	8	8	9	9	10	
Key4	11	11	12	12	13	14	
Key5	15	15	15	16	16	17	17 17

***Table I.5**. Synonym codes syn_id table.*

where each cell contains the code associated to the synonym (*syn_id*) in Table I.4.

An important control to be done during the execution of the algorithm concerns the correct ordering of the sequence of atoms progressively identified. To easy the access to such information the following table of position should be instantiated:

Key1	1	2	1	2	3	1	2
Key2	1	2	1	2	3	1	2
Key3	1	1	2	1	2	1	
Key4	1	2	1	2	1	1	
Key5	1	2	3	1	2	1	2 3

***Table I.6**. Table of positions.*

where the position of each atom within its own synonym is recorded.

2.2.3 Check point

Input:

- Key and synonym files (see Table I.1):
  \check_point\preprocess\syns_Claim.csv

Output:

- Synonyms split in words (see Table I.2):
  \check_point\preprocess\atoms_key.csv
- Synonym information table (see Table I.4):
  \check_point\preprocess\atom_code_Figure.csv
- Synonyms codes table (see Table I.5): *\check_point\preprocess\atoms_code.csv*
- Table of positions (see Table I.6):
  \check_point\preprocess\position_synonimus.csv

2.3 Phase 2: key search

The proposed methodology proves indeed very effective; nevertheless, it hides some complexity mainly for the various validation controls that the candidate solutions must pass in order to be stored in the final report. To highlight the details of the method, let's start again from the simple document in Example I.1. Hence, we will show a clear (although incomplete) guideline useful to comprehend all the steps needed for the final report generation.

2.3.1 A practical example

Let us consider again the artificial short document shown in Example I.1, which we report below for convenience.

DOC: "your contract ref: pami of new mexico . participation: 19.50% of 100% our reference: aon ben date of loss: 12/07/2010"

***Example I.1**. Document used as leading example.*

As already mentioned, the text is firstly split into single words (*text words* or, when clear from the context, simply *words*), using as splitting delimiters both the space character (single and multiple) and the colon mark “:”. As will be clear from the following sections, it is useful to prepend an astherisk “*” to all the words preceded by a colon. In fact, we recognized that values separated by the corresponding key through a colon mark contain usually a meaningful information; thus, prepending an asterisk to such values insures us about their relevance. As a result, the text in Example I.1 will keep the following form:

t	terms
1	your
2	contract
3	ref
4	*pami
5	of
6	new
7	mexico.
8	participation
9	*19.50%
10	of
11	100%
12	our
13	reference
14	*aon
15	ben
16	date
17	of
18	loss
19	*12/07/2010

Table I.7. *Text words constituting the document in Example I.1. We prepend an asterisk to words preceded by a colon.*

Let us now proceed step by step to the document scan.

t = 1 → **“your”**

We are called to measure the distance of the first term from each atom listed in Table I.2. Having specified a tolerance parameter θ, we select all those atoms having distance ≤ θ, so that all the keys associated to the corresponding synonyms, raised to the role of candidate solutions, will be inserted into the *stack of solutions* data structure. As distance function we decided to use the “Levenshtein distance”, a typical edit distance whose properties may be found in the following web site: http://en.wikipedia.org/wiki/Levenshtein_distance. Fixing θ = 1 and having read the first word “your”, we obtain the following table of distances

Key1	**0**	6	**0**	6	4	**0**	8	
Key2	**1**	6	**1**	6	4	**1**	8	
Key3	6	6	4	4	4	6		
Key4	**0**	4	6	4	13	13		
Key5	4	3	3	3	4	4	3	5

Table I.8. *Table of distances between the current text word “your” and the atoms listed in Table I.2.*

where each cell contains the distance between the current text word and the atoms listed in Table I.2. To go back to the selected candidate keys, we make use of the *syn_id* table (see Table I.5), containing all the synonyms code, and the synonym information table (see Table I.4), where the key-synonym correspondence is stored. From the former we recognize that synonyms having *syn_id* equal to 1, 2, 3, 4, 5, 6 and 11 satisfy the similarity constraint; from the latter we may trace back to the corresponding *key_ids*, hence keys: key1 (for *syn_id* 1, 2, 3), key2 (for *syn_id* 4, 5, 6), and key 4 (for *syn_id* 11). In detail:

- *syn_id* 1 (“your contract”) → key1
- *syn_id* 2 (“your contract ref”) → key1
- *syn_id* 3 (“your reference”) → key1
- *syn_id* 4 (“our contract”) → key2
- *syn_id* 5 (“our contract ref”) → key2
- *syn_id* 6 (“our reference”) → key2
- *syn_id* 11 (“your” share) → key4

Among them we recognize, by simple inspection of the document to be processed, that the correct solution is identified by the synonym having *syn_id* 2. However, we are able to validate it only at time *t* = 3 after having read the text word “ref”. This is why we need the *stack of (candidate) solutions* data structure, allowing us to keep track of all the candidate solutions and especially of those which will be validated, and therefore inserted in the final report. Namely, we fill the following table:

syn_id	key_id	count	valid	active	value	sum of distances
1	1	1	0	0		0
2	1	2	0	0		0
3	1	1	0	0		0
4	2	1	0	0		1
5	2	2	0	0		1
6	2	1	0	0		1
11	4	1	0	0		0

Table I.9*. Stack of candidate solutions at t = 1.*

where:

- **count**: initially it is the number of atoms composing the matching synonym (see *syn_length* in Table I.4). Hence, at each successive *matching iteration* (i.e. each time a subsequent atom of the same synonym matches the current text) this number decreases by 1 unit. In other words, *count* represents the number of matching atoms we need in order the candidate solution to become valid;
- **valid**: a solution is valid when it has passed all the validation checks (e.g. *count* reaches zero and atoms ordering has been respected);
- **active**: tells us that the solution is valid and the recording of the corresponding value is active;
- **value**: the current value of the solution which typically corresponds to the words following a valid solution until a stopping criterion is reached;

- **sum of distances**: is the sum of the distances between the single atoms and the individual text words; it allows us to identify what is the best solution in case multiple keys are activated.

Before decreeing a new solution as valid, we must check whether its position is correct, i.e. the first text word matches with the first atom. Thanks to the *table of positions* (see Table I.6) we may easily check that all our candidate solution start from position 1. By using the The *position control* is always performed even on existing candidate solutions to make sure that the atoms of these solutions are placed in the correct order; otherwise the solution will be removed from the stack.

Let us now pass to the second text word:

t = 2 →“your **contract**”

The following events occur:

- *syn_id* 1 (“your contract”) → key1 valid
- *syn_id* 2 (“your contract ref”) → key1 requires further processing
- *syn_id* 3 (“your ~~reference~~”) → key1 discarded as no longer matches the atom
- *syn_id* 4 (“our contract”) → key2 valid
- *syn_id* 5 (“our contract ref”) → key2 requires further processing
- *syn_id* 6 (“our ~~reference~~”) → key2 discarded as no longer matches the atom
- *syn_id* 7 (“contract”) → key3 added and immediately validated being the synonym composed of one atom
- *syn_id* 8 (“contract name”) → key3 added
- *syn_id* 11 (“your ~~share~~”) → key4 discarded as no longer matches the atom

Note that only the synonyms whose first atom matches the current text word “contract” are added as candidate solutions, according to the aforementioned *position control*.

The *stack of candidate solution* will be updated as follows:

syn_id	key_id	count	valid	active	value	sum of distances
1	1	0	1	0		0
2	1	1	0	0		0
4	2	0	1	0		1
5	2	1	0	0		1
7	3	0	1	0		0
8	3	1	0	0		0

Table I.10. Stack of candidate solutions at *t* = 2. The cell modified from the last step have been highlighted in light blue.

Note that, despite some solutions have become valid, they are still deactivated, and will enter in the active state only in the next iteration.

Third step:

t = 3 →“your contract **ref**”

Below are the events occurred:

- *syn_id* 1 (“your contract **ref**”) → key1 becomes active: “ref” is saved as part of the value
- *syn_id* 2 (“your contract ref”) → key1 valid
- *syn_id* 4 (“our contract **ref**”) → key2 becomes active: : “ref” is saved as part of the value
- *syn_id* 5 (“our contract ref”) → key2 valid (note that it will be discarded in a later step, having distance higher than *syn_id* 2)
- *syn_id* 7 (“contract **ref**”) → key3 becomes active: : “ref” is saved as part of the value
- *syn_id* 8 (“contract ~~name~~”) → key3 discarded as no longer matches the atom

The *stack of candidate solution* will be updated as follows:

syn_id	key_id	count	valid	active	value	sum of distances
1	1	-1	1	1	ref	0
2	1	0	1	0		0
4	2	-1	1	1	ref	1
5	2	0	1	0		1
7	3	-1	1	1	ref	0

Table I.11. Stack of candidate solutions at *t* = 3. Highlighted cells: light blue: modified from the last step; light red: superseded by better solutions.

When more than one key is in the *active* state, further specific *cleaning controls* are performed: i) solutions whose synonym has fewer atoms than other active solutions will be discarded (they are subset of the latter) as *syn_id* 7 (“contract”) when compared to *syn_id* 2 (“your contract”); ii) whenever various active solutions have the same synonym length, only those having minimum distance will be retained, as *syn_id* 1 w.r.t. *syn_id* 4; iii) in case two solutions share the same synonym length and distances, both will be kept and their value field will start to be filled. Thus, the *stack of candidate solution* becomes:

syn_id	key_id	count	valid	active	value	sum of distances
1	1	-1	1	1	ref	0
2	1	0	1	0		0
5	2	0	1	0		1

Table I.12. Stack of candidate solutions at *t* = 3, when superseded cells have been removed.

Moreover, the presence of active keys calls for a stopping rule which, once satisfied, allows us to free the corresponding entry from the stack of solutions, and store the extracted value in the final report as well. The stopping criteria taken into account are as follows:

1. reached maximum number of words for the key (for example 10);

2. presence of a punctuation mark in the value (such as a period at the end of the word);
3. activation of a new key;
4. presence in the value of one of the stop-words listed in a input file (this policy is disabled for now);
5. end of the document.

In the leading example, no stop criteria has been reached.

Proceeding with the document scan:

t = 3 →“your contract ref **pami**”

The new word is not similar to any atoms listed in Table I.2, so no new solution is activated in this step and those not yet decreed as valid must be deleted from the stack (this is not the case in our example).

Below are the events occurred:

- *syn_id* 1 (“your contract **ref pami**”) → key1 active: “pami” is added to the value
- *syn_id* 2 (“your contract ref **pami**”) → key1 becomes active: “pami” is saved as part of the value”
- *syn_id* 5 (“our contract ref **pami**”) → key2 becomes active: “pami” is saved as part of the value”

The *stack of candidate solution* reads now:

syn_id	key_id	count	valid	active	value	sum of distances
1	1	-2	1	1	ref pami	0
2	1	-1	1	1	pami	0
5	2	-1	1	1	pami	1

Table I.13. Stack of candidate solutions at *t* = 4. Same notation as Table I.11.

Again we are in a situation where three keys are in the *active* state, so that the cleaning controls must be performed. As usual, we retain those solutions whose keys has minimum distance, or greater synonym length, or both: as a consequence *syn_id* 1 and *syn*_id 5 will be removed, obtaining the following stack:

syn_id	key_id	count	valid	active	value	sum of distances
2	1	-1	1	1	pami	0

Table I.14. Stack of candidate solutions at *t* = 4, when superseded cells have been removed.

In this case no stopping criteria has been reached. In particular, the method loops untile time *t* = 7 with the word “Mexico”: in fact we find a punctuation mark, a point “.”, decreeing the value as completed. It will be saved in the final report and *syn_id* 2 will be released, freeing an entry in the stack.

Remark. If the word “mexico.” were not followed by the symbol “.”, *syn_id* 2 would have been saved only at time *t* = 8, where the word “participation” would have indicate the activation of *syn_id* 4. In this case, a further trick leads us to remove the string “participation” from the value of *syn_id* 2, that was saved due to the sequential behavior of the method.

So far we have followed a simple case study showing how the stack of solutions should be updated. In the next section, we will detail the whole procedure to be followed in order to prepare the final report.

2.3.2 Summing up

The proposed procedure takes in input a set of documents: doc_1, doc_2, …, which will be processed one by one through an iterative procedure scanning the single text words. Figure I.1 shows the main steps of the method.

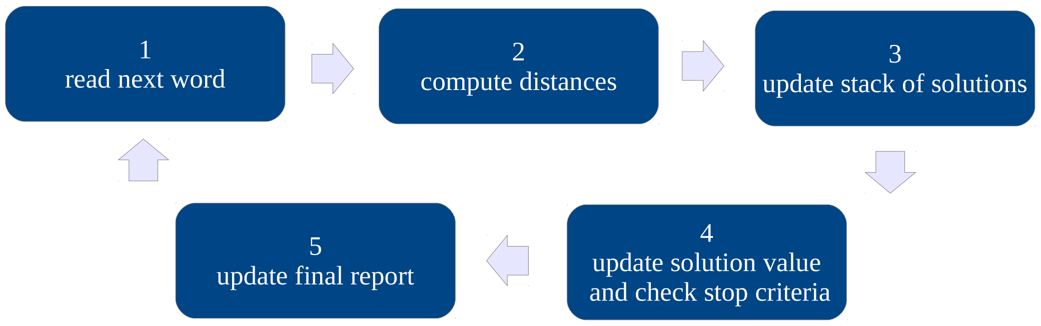


Figure I.1*. Iterative procedures for key-values search.*

2.3.2.1 General parameters

Given in input:

- a set of documents to be processed: doc_1, doc_2, …;
- *tolerance* threshold θ (default θ = 1): parameter needed to define when two words are similar each other (*d*(word1,word2) ≤ θ);
- *num_max_words_value* parameter: max number of words a value may be composed of; when the maximum is reached, all subsequent strings are discarded;
- *no_errors_length_permission* parameter (default = 2): this parameter forces an exact match (distance = 0) in case of words having length ≤ *no_errors_ length_permission*. We introduce this threshold because no partial matches are contemplated for very short words, such as “id” or “fd”; it has been experimentally observed that this trick improves the performance of the algorithm;

computes the table of text words constituting doc_*i*, for each document (see Table I.7).

Check points

input: *i*-th document (see Example I.1): *..\dataset\..*

output: text words list (see Table I.7):
\check_point\docs_check\dataset\name_doc_file\atoms_doc.csv

In the following, we will describe the various steps depicted in Figure I.1. For each document doc_*i*, the following procedures must be executed in the same order as they appear here below.

2.3.2.2 Procedure 1: read next text word

In input the table of text words constituting the document to be processed (see Table I.7) together with the iteration counter *t*, read the next text word to be processed, according to the following instruction:

current_word = read the (t+1)-th word in Table I.7.

2.3.2.3 Procedure 2: compute distances and select candidate solutions

The aim of this procedure is to compute the distance between the *current_word* and each atom listed in the table of atoms. The following table of distances will be generated:

d(*current_word*, *atom*_1)	*d*(*current_word*, *atom*_2)			
		d(*current_word*, *atom*_*i*)		

***Table I.15**. Table of distances between the current text word and the atoms listed in the table of atoms (similar to Table I.2).*

where each cell stores the Levenshtein distance between *current_word* and the atom the cell corresponds to. Note that, w.r.t. Table I.8 where each row contains all atoms referring to the same key, here we list all atoms in sequence. The correspondence between atoms and related keys will be restored thanks to the synonym information table and the *syn_id* table (see Table I.5 and I.4, respectively).

From this table, the list of candidate solutions (*candidate_synonyms*) is now generated, containing all synonyms associated to atoms having distance *d*(*current_word*, *atom*) $\leq \theta$. Note that in case length(*current_word*) ≤ *no_errors_ length_permission*, *d*(*current_word*, *atom*) should be exactly equal to 0.

The list of *candidate_synonyms* = (syn_1, syn_2, …) will be populated, containing all those synonyms most similar to the *current_word*.

Check points

input: text words list (see Table I.7):
\check_point\docs_check\dataset\name_doc_file\atoms_doc.csv

output: table of distances (see Table I.15):
\check_point\docs_check\dataset\name_doc_file\distances.csv

2.3.2.4 Procedure 3: update the stack of candidate solutions

This procedure, in input the candidate solutions (*candidate_synonyms* list) generated in the previous step, updates the stack of solutions (see Table I.9 et seq.) by (possibly): i) adding new candidate solutions, ii) removing those which are nomore valid, iii) modifying the state and the value of the solutions listed in the stack, and iv) performing some further check should multiple active solutions be present. In the following, we will concentrate on the various typologies of operations to be performed on the stack. Again, such procedures should be executed exactly in the same order as they are explained here below.

1. **Insertion of new candidate solutions**

```
foreach candidate solution i not yet stored in the stack
    let x contains the coordinate of the atom whose distance from the current_word is
        under analysis (see Table I.15)
    let pos contains the value of x-th cell in the table of positions (see Table I.6)
    // if the synonym start from the wrong position, then remove it
    if pos ≠ 1 then remove the synonym from the list of candidate_synonyms
        else add a row to the stack of solutions (Table I.9) filled with all
            information related to the added synonym (retrieved in synonym
            information table and the syn_id table – see Table I.5 and I.4,
            respectively)
```

syn_id	key_id	count	valid	active	value	sum of distances
...	...	...	...	...	...	...
...	...	...	...	...	...	...
new syn	new key	length – 1	0	0		*d*(*current_word*, *atom_i*)

```
    endif
endfor
```

Algorithm I.1. *Pseudocode of the insertion of new candidate solution in the stack.*

2. **Deletion of incorrect candidate solutions**

```
foreach candidate solution i in the stack
   // check correctness of atoms constituting the synonym
  if ((active = 0) and (valid = 0) and (syn_id ∉ candidate_synonyms))
         then remove the solution from the stack
   endif
endfor
```

Algorithm I.2*. Pseudocode of the deletion of incorrect candidate solutions.*

3. **Update of the (not yet valid) candidate solution flags and deletion of solutions not respecting the correct text words ordering**

```
foreach candidate solution i in the stack
   // only for solutions already listed in the stack at time t – 1 and not yet valid
  if ((count ≠ length – 1) and (valid = 0)) then
         count = count – 1                  // decrease the count
         sum_dist = sum_dist + d(current_word, atom_i)  // update sum of distance field
         // check that the new word has a correct position
         let pos be defined as in Algorithm I.1
         if (pos  ≠ length – count) then remove solution endif
         // tag as valid the solutions having count = 0 (completed match)
         if (count = 0) then valid = 1 endif
         //tag as active the solutions with count<0 (tagged as valid in the previous step)
         if (count < 0) then active= 1 endif
   endif
endfor
```

Algorithm I.3*. Pseudocode of the management of candidate solutions.*

4. **Check of multiple active solutions**

   More than one solution is often active in the same iteration. In most cases this occurs when a key is active that is a substring of another longer active solution. Sometimes, we may be in the situation wherein an active solution has not yet reached the stopping criterion and at the same time a new solution has been found. In this case, the old solution will be saved in the final report and removed from the stack as well, as the new atom found is considered as part of a new candidate solution.

```
// if more than an active solution exists in the stack
if number_active_solutions > 1 then
   // initialize an empty list containing those solutions to be deleted
   set_delete_solutions = {}                    // empty list
```

```
// retain only those solutions related to different keys having both the same length
// and minimum distance (it may happens that more keys become active: some with
// null distance, some others with distance ≤ tolerance θ * synonym_length;
// keep only those keys having minimum distance

// search in the stack those solutions related to different keys having the same length
set_solutions = select_solutions (same_length and different_key_reference)

// search the solution having minimum distance, saving the latter in dist_min
dist_min = find_min_distance_sol (set_solutions)

// add to the list of solutions to be deleted those having distance > dist_min

set_delete_solutions = find_sol_with_dist_greater_than (dist_min)

// retain only one solution among those related to the same key having the same
// length (in case more synonyms exist of a same key with distance ≤ tolerance θ, all
// enters into the active state)

// search in the stack those solutions related to the same key having the same length
set_solutions = select_solutions (same_length and same_key_reference)

// take only one solution and add the others to the list of solutions to be deleted
set_delete_solution = take_only_one (set_solutions)

// at this point there will be at most two active solutions. In such case the following
// control must be performed:
// case 1: contiguous match
```

t – 5	***t – 4***	***t – 3***	***t – 2***	***t – 1***	***t***
			sol_1	sol_1	sol_1
			sol_2	sol_2	

```
// sol_2 is a subset of sol_1 and it will be deleted.
// case 2: non contiguous match
```

t – 5	***t – 4***	***t – 3***	***t – 2***	***t – 1***	***t***
			sol_1	sol_1	sol_1
sol_2	sol_2				

```
// sol_2 will be send to the final report and then removed from the stack

if (size (active_sol) > 1) then
    count_1 = get_count_solution_1
    count_2 = get_count_solution_2
    if (count_1 < count_2) then
        new_sol = active_sol (2)
        old_sol = active_sol (1)
    else
        new_sol = active_sol (1)
```

```
            old_sol = active_sol (2)
        endif

        lenght_new = get_lenght (new_sol)
        lenght_old = get_lenght (old_sol)

        count_new = get_count (new_sol)
        count_old = get_count (old_sol)

        // if old_key is a subset of new_key, then delete old_key, else save old_key
        if (lenght_new ≥ (abs(count_old) + lenght_old – 1)) then
            // delete old solution
            set_delete_solutions = append (old_sol)

        else
            // save old solution and delete new_key from its value
            send_solutions = append (new_sol)
            set_delete_solutions = append (new_sol)
        endif
    endif
endif
```

***Algorithm I.4**. Pseudocode of the management of multiple active solutions.*

Check point

Input: all files generated till now;

Output: updated stack of solutions (see Table I.9):
\check_point\docs_check\dataset\name_doc_file\solution_stack.csv

2.3.2.5 Procedure 4: update active solutions values and check stop criteria

In this step the value of all active solutions is updated and the control on the stopping criteria is launched.

```
add current_word to the value of each active solution in the stack
```

syn_id	key_id	count	valid	active	value	sum of distances
...	...	...	...	...	...	...
...	...	...	...	...	...	...
syn_id	*key_id*	*count*	1	1	*old_value* + *current_word*	*d*(*current_word*, *atom_i*)

```
foreach active solution act_sol in the stack
    // condition 1: reached maximum number of words per key)
    let cond_1: (number_words (value) = num_max_words_value)
    // condition 2: presence of a punctuation mark in the value
    let cond_2: (string_match (value, punctuation_mark))
    // condition 3: presence in the value of at least one of the stop-words provided in
    // input (disable in the current release)
    let cond_3: (string_match (value, stop-words))
    // condition 4: end of document
    let cond_4: (t = numer_words_document)

    if (cond_1 or cond_2 or cond_3 or cond_4) then
        // insert the solution to the send_solutions list (solutions to be inserted in the
        // final report
        send_solutions = append (solution)
    endif
endfor
```

Algorithm I.5. *Pseudocode of the value update and stop criteria check.*

2.3.2.6 Procedure 5: final report update

This step is responsible for saving in the final report any complete solution by inserting it in the *send_solutions* list. If the key of the new solution to be inserted in the report is already present therein, i.e. another occurrence of the same key has been found and, together with its value, both have formerly been inserted in the report, then add a new session in the report. Table I.16 shows a prototypical report file.

		key1			**key i**			**key n**		
doc	**session**	**value**	**generated_ from**	**position**	**value**	**generated_ from**	**position**	**value**	**generated_ from**	**position**
doc_1	1			2						1
	2			1						2
doc_4	1			1						3
	2			2						1

***Table I.16**. Final report.*

Each column contains, respectively: the document path, the session number, and, for each key: the extracted value, the synonym prefixing the value, and the position, i.e. a counter (whose scope is the single session) providing the temporal order along which the keys have been encountered.

Check points

input: all files generated till now;

output: updated report (see Table I.16):
\check_point\docs_check\dataset\name_doc_file\final_report.csv

2.3.2.7 Final step

Having reached the end of the last document, the final report (see Table I.16) will be exported in “csv” format.

Check points

input: all files generated till now;

output: report in “csv” format (see Table I.16): *\results\output.csv*

3 PART II

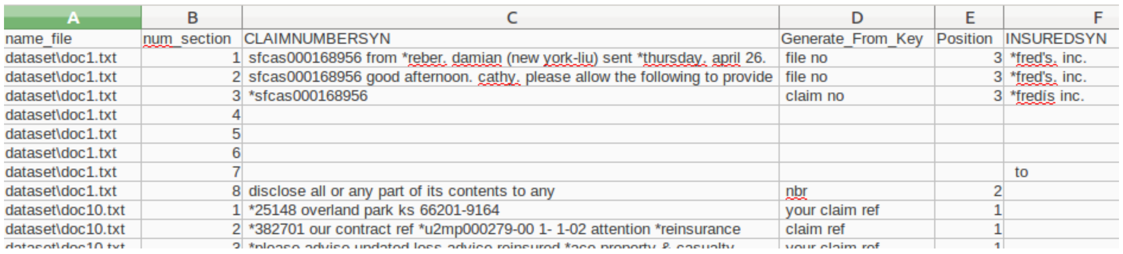
3.1 Phase 1: one file per key

3.1.1 Description

The objective of this phase is to extract from the file produced in Part I a set of files each containing all information pertaining a single key. Moreover, a first preprocessing is performed while creating the above files, consisting in retaining only those records which prove to be meaningful in the subsequent steps. In detail, as each document may be composed by various sessions, each key may appear different times: sometimes these occurrences are all significant, some others they contain redundant information or even dirtiness. As in Part I we recognized that whenever a key is followed by a colon, the corresponding value is usually significant, we adopted the heuristic consisting of inserting an asterisk (*) before the value when the key ends with a colon. Therefore, for fixed document, for each key we retain: i) only those records preceded by an asterisk, in case at least a record has one *, or ii) all records, in case no asterisk is present as first element of each field.

3.1.2 Implementation

The output file of Part I (*output.csv*) contains a survey of the information collected up to now (see Figure II.1). While *name_file* and *num_session* columns explain which session of which document the keys refer to, *name_key* ∊ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}, containing the values associated to the reference key, is followed by the fields g*enerated_from* (indicating which synonym of the key precedes the value) and p*osition* (specifying, within each session, the temporal order along which the keys are discovered).



A	B	C	D	E	F
name_file	num_section	CLAIMNUMBERSYN	Generate_From_Key	Position	INSUREDSYN
dataset\doc1.txt	1	sfcas000168956 from *reber. damian (new york-liu) sent *thursday. april 26.	file no	3	*fred's. inc.
dataset\doc1.txt	2	sfcas000168956 good afternoon. cathy. please allow the following to provide	file no	3	*fred's. inc.
dataset\doc1.txt	3	*sfcas000168956	claim no	3	*fredis inc.
dataset\doc1.txt	4				
dataset\doc1.txt	5				
dataset\doc1.txt	6				
dataset\doc1.txt	7				to
dataset\doc1.txt	8	disclose all or any part of its contents to any	nbr	2	
dataset\doc10.txt	1	*25148 overland park ks 66201-9164	your claim ref	1	
dataset\doc10.txt	2	*382701 our contract ref *u2mp000279-00 1- 1-02 attention *reinsurance	claim ref	1	
dataset\doc10.txt	3	*please advise updated loss advice reinsured *ace property & casualty	your claim ref	1	

Figure II.1. *Content of output.csv.*

The files we are going to generate will refer to the single keys, and in particular will contain the following fields: *doc_id* (document file name cleaned by the path information), *key_value* (the only significant values of the considered key), *generated_from* and *position* (as before). The session information is implicit in the number of rows referring to the same document.

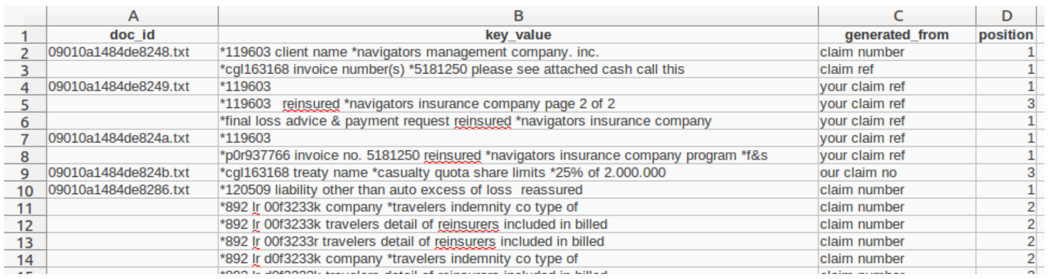
In order to retrieve only the meaningful values, a check on the presence of an asterisk at the beginning of each value must be done. Namely, the pseudocode of this filter is reported in Algorithm II.1.

```
for each document d
    let n be the number of records r referring to the same document d
    for each record r referring to the same document d
        if the corresponding value field begin with * then keep it
                                                   else mark the value to be deleted
        endif
    if the total number of values to be deleted equals n then remove all marks from all records r
    endif
    erase marked values
endfor
```

Algorithm II.1. *Pseudocode of the filter retaining only meaningful values.*

To avoid problems with the comma delimiter of the *csv* files, from now on we will assume that all commas have been translated in semicolons.

Figure II.2 shows the content of a prototypical file generated by the above routine.



	A	B	C	D
1	**doc_id**	**key_value**	**generated_from**	**position**
2	09010a1484de8248.txt	*119603 client name *navigators management company. inc.	claim number	1
3		*cgl163168 invoice number(s) *5181250 please see attached cash call this	claim ref	1
4	09010a1484de8249.txt	*119603	your claim ref	1
5		*119603 reinsured *navigators insurance company page 2 of 2	your claim ref	3
6		*final loss advice & payment request reinsured *navigators insurance company	your claim ref	1
7	09010a1484de824a.txt	*119603	your claim ref	1
8		*p0r937766 invoice no. 5181250 reinsured *navigators insurance company program *f&s	your claim ref	1
9	09010a1484de824b.txt	*cgl163168 treaty name *casualty quota share limits *25% of 2.000.000	our claim no	3
10	09010a1484de8286.txt	*120509 liability other than auto excess of loss reassured	claim number	1
11		*892 lr 00f3233k company *travelers indemnity co type of	claim number	2
12		*892 lr 00f3233k travelers detail of reinsurers included in billed	claim number	2
13		*892 lr 00f3233r travelers detail of reinsurers included in billed	claim number	2
14		*892 lr d0f3233k company *travelers indemnity co type of	claim number	2

Figure II.2. *Content of a prototypical file output_CLAIMNUMBERSYN.csv.*

3.1.3 Check points

Input: file *\check_point\post_process\output.csv*

Output: files *\check_point\post_process\output_name_key.csv*, where *name_key* ∈ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}.

3.2 Phase 2: filtering and cleaning

3.2.1 Description

Each file generated in Phase 1 contains as values both significant and redundant (or even needless) information. To retain only those values which are really relevant, we propose a 3-step procedure aimed at filtering and cleaning the various records. The implementation of these procedures is key-based, in the sense that each key has its own methodology for filtering and cleaning. In detail, for each key, the three steps are defined as follows:

1. **filtering by synonyms**: given a set *bad_syn* of synonyms, discard all values that have been generated from (i.e. whose field *generated_from* contains) a synonym in *bad_syn*;
2. **filtering by regex**: given a set *filter_regex_set* of regular expressions, remove all values matching any of them. Note that some regular expressions may be easily introduced in the code through alternative methods, on the basis of the facilities of the chosen programming language. Moreover, the same regular expressions may have a different syntax from one language to another, mainly for subtle issues such as the escape character or other notational symbols. In the sequel, both the goal each regex tries to achieves and its implementation in Mathematica will be provided, with the exception of expressions which, for the sake of simplicity, may be defined out of the regex framework;
3. **cleaning by regex**: given a list *clean_regex_set* of regular expressions, apply its elements respecting the order whereby they are list, obtaining as output an iterative refinement of the values.

The next section will describe in detail the three steps for each assigned keys. Note that a *trim* routine should be applied to all values in order to remove all leading and trailing occurrences of space character.

3.2.2 Implementation

3.2.2.1 Regex list

The following Table II.1 contains a list of the employed regular expressions, providing: the *id* of the regex, used to simplify subsequent recalls, the *name* we associated to the regex, the Mathematica *code*, a brief *description* of its functioning, and a couple of *examples*. The latter will have the form *str* → *output*, where *str* is the input string (usually the field *value*), and output is the result of the application of the *regex* to *str*. The Mathematica code to obtain *output* starting from *str* is as follows:

```
StringCases[str, RegularExpression[regex]] = output
```

The empty string is denoted with “ ”.

Sometimes finding the most appropriate regular expression with a given behavior may be much more complicated than using alternative Mathematica routines (possibly involving themselves regular expressions). In such cases the *Code* field will contain the name of the Mathematica routine used for that purpose (whose definition may be found in the attached Mathematica notebook `post_filtering.nb`).

Id	Name	Code	Description	Example
Filtering regex				
1	*delete_if_no_number*	.*[\\d]+.*	Delete *str* if it contains no number	"demo"→" " "demo 4"→ *"demo 4"*
2	*delete_if_begin_with*	`deleteIfBeginWith[str, list]`	Delete *str* if it begins with a word belonging to the set *list*[2].	let {"home", "house"} ∈ *list* "my home is"→"my home is" "home sweet" → ""
3	*delete_if_end_with*	`deleteIfEndWith[str, list]`	Delete *str* if it ends with a word belonging to the set *list*[2].	let {"home", "house"} ∈ *list* "my home is"→"my home is" "sweet home" → ""
4	*delete_if_contain*	`deleteIfContain[str, list]`	Delete *str* if it contains (in any position) at least one of the words belonging to the set *list*[2].	let {"home", "house"} ∈ *list* "my home is"→"" "sweet homework" → "sweet homework"
5	*date_cleanup*	`dateCleanup[date]`	Retain only those dates in the right format[2]. For instance remove date having 0 in place of a correct day, month or year	"0/01/2012" → "" "01/01/2012" → "01/01/2012"
Cleaning regex				
6	*clean_first_asterisk*	(?!^*)\\w.*	Erase the first asterisk at the beginning of the string, eventually preceded by empty spaces, retaining all the remainder	" * demo *"→"demo *" "d * demo* 4"→"d * demo* 4" "demo"→"demo"
7	*take_till_first_asterisk*	^([^*]+)	Take all characters before the first asterisk, erasing what follows (asterisk included)	"proof * demo * last"→"proof " "no asterisk"→"no asterisk"
8	*take_till_no_number*	(^(([\\w\\\\/\\-()]+)?[\\d-]?(\\w+)?)([.-/\\\\]+([\\w]+)?[\\d-](\\w+)?)*(?=[ ]+[^0-9]+([\\s]?)|([^0-9])|[0-9]?))	Take all alphanumeric characters from the beginning, removing all characters following the first word without numbers (included). In any case it retains the first word.	"in this3 *case what"→"in this3" "instead in this case what"→"instead" "33 22 a3 lot3 of numbers"→"33 22 a3

[2] Note that each word in *list* may itself be a regular expression. Please pay attention to the spaces before and after each word, when present.

Id	Name	Code	Description	Example
				lot3”
9	*take_till_number*	^(([^0-9]*)[]((?=((([\\w]+)?[\\d\\-\\\\\//\\$](\\w+)?)))|.*)|([^0-9]*))	Take all literal characters from the beginning, removing all characters following the first word containing numbers (included). If the first word contains numbers, then it retains only the letters preceding the first number.	“in this3 *case what”→“in ” “instead in this case what”→“instead in this case what” “instead in this3 case what”→“instead in ” “instead3 in this case what”→“instead” “33 22 a3 lot3 of numbers”→“”
10	*delete_what_follows*	`deleteWhatFollows[str, list]`	Take the initial portion of *str* until a match with a word in *list* occurs. The matched word is removed[2].	let {“home”, “house”} ∈ *list* “sweet home alabama”→ “sweet ” “clean string” →“clean string”
11	*delete_what_follows_keeping*	`deleteWhatFollowsKeeping[str, list]`	Take the initial portion of *str* until a match with a word in *list* occurs. The matched word is retained[2].	let {“home”, “house”} ∈ *list* “sweet home alabama”→ “sweet home” “clean string” →“clean string”
12	*delete_match*	`deleteMatch[str, list]`	Take *str* as is, after having removed all the characters matching any of the words in *list*[2].	let {“home”, “house”} ∈ *list* “sweet home alabama”→ “sweet alabama” “clean string” →“clean string”
13	*date_match_1*	\\d{1,2}(\\,|\\.|\\/|\\-)+\\d{1,2}(\\,|\\.|\\/|\\-)+(\\d{4}|\\d{2})	One of the possible format a correct date must respect	“*jan 01 08 - dec 31 08” → “” “*12/1/1972 location” → “12/1/1972” “*01-oct-2005 - 30-sep-2006 your share .. 10.00000%” → “”
14	*date_match_2*	\\d{1,2}(\\,|\\.|\\s|\\-)+\\d{0,4}(\\,|\\.|\\s|\\-)+(january|february|march|april|may|june|july|august|september|october|november|december|jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)	One of the possible format a correct date must respect	“*jan 01 08 - dec 31 08” → “01 08 – dec” “*12/1/1972 location” → “” “*01-oct-2005 - 30-sep-2006 your share .. 10.00000%” →

Id	Name	Code	Description	Example
				“05 - 30-sep”
15	*date_match_3*	(january|february|march|april|may|june|july|august|september|october|november|december|jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)(\\,|\\.|\\s|\\-)+\\d{1,2}\\s*(st|nd|rd|th){0,1}\\s*(\\,|\\.|\\s|\\-)+\\d{0,4}	One of the possible format a correct date must respect	“*jan 01 08 - dec 31 08” → {“jan 01 08”, “dec 31 08”} “*12/1/1972 location” → “” “*01-oct-2005 - 30-sep-2006 your share .. 10.00000%” → “”
16	*date_match_4*	\\d{1,2}(\\,|\\.|\\s|\\-)*\\s*(st|nd|rd|th){0,1}\\s*(january|february|march|april|may|june|july|august|september|october|november|december|jan|feb|mar|apr|may|jun|jul|aug|sep|oct|nov|dec)(\\,|\\.|\\s|\\-)+\\d{0,4}	One of the possible format a correct date must respect	“*jan 01 08 - dec 31 08” → “08 - dec 31” *12/1/1972 location → “” “*01-oct-2005 - 30-sep-2006 your share .. 10.00000%” → {“01-oct-2005”, “30-sep-2006”}
17	*to_standard_date*	`toStandardDate[date]`	transform *date* in the “dd/mm/yyyy” date format	“03/01/2012” → “01/03/2012” “march 6th 2012” → “06/03/2012”
18	*date_cat*	`dateCat[datelist]`	take all dates in *datelist* and join them in a single string using “ - ” as separator	{“march 6th 2012”, “february 3rm 2011”} → “march 6th 2012 - february 3rm 2011” {“march 6th 2012”}→ “march 6th 2012”

19	*claimnumber_match_1*	((\\$*)\\d(\\%*)(\\/*)(((\\w*)(\\$*)(\\%*)(\\/*)(\\s*)(\\-*)(\\.*))*)){1,}	This regex is applied on a value (*<u>str</u>*) after it has been splitted using the space character as delimiter. It maintains only strings respecting the format (a sort of code with digits, literals and other symbols in a given position) specified by the regex. For more information see `claimnumberMatch1[str]`	“l01036894 (m) invoice number(s) *5180918 please see attached. this message” → {{"01036894"}, {}, {}, {}, {"5180918"}, {}, {}, {}, {}, {}}
				“*012028 our contract ref *u2k80003 80-001-1-03 attention *reinsurance” → {{"012028"}, {}, {}, {}, {"2k80003"}, {"80-001-1-03"}, {}, {}}

20	*claimnumber_match_2*	((((\\w*)(\\s*)(\\$*)(\\%*)(\\/*)(\\-*)(\\.*))*)(\\$*)\\d(\\%*)(\\/*)){1,}	Similar to 19, for a different ordering of the code. For more information see `claimnumberMatch2[str]`	“l01036894 (m) invoice number(s) *5180918 please see attached. this message” → {{"101036894"}, {}, {}, {}, {}, {"5180918"}, {}, {}, {}, {}, {}}

Id	Name	Code	Description	Example
				"*012028 our contract ref *u2k80003 80-001-1-03 attention *reinsurance" → {{"012028"}, {}, {}, {}, {"u2k80003"}, {"80-001-1-03"}, {}, {}}

***Table II.1**. Regex List used in the filtering and cleaning process.*

In the next five sections, we will sketch the operations to be performed on the values related to the same key. We will follow the general schema introduced in Section 3.2.1, listing whenever pertinent, the subroutine to be followed in order to complete the filtering/cleaning process, referring mainly to the ones reported in Table II.1. Wherever not expressly indicated, the temporal order along which the subroutines have been listed should be respected.

3.2.2.2 POLICYSYN

1. **filtering by synonyms**: none
2. **filtering by regex**:
   - *1. delete_if_no_number*
3. **cleaning by regex**:
   - *6. clean_first_asterisk*
   - *7. take_till_first_asterisk*
   - *8. take_till_no_number*

3.2.2.3 CLAIMANTSYN

1. **filtering by synonyms**:
   - discard the *value* if *generated_from* ∊ {"claimant", "claimant name", "claimants", "plaintiff"} and *value* does not begin with an asterisk
2. **filtering by regex**:
   - *2. delete_if_begin_with,* where *list* ={"type ", "in ", "line ", "building ", "product ", "can ", "to ", "current ", "the purpose ", "and "}
3. **cleaning by regex**:
   - *6. clean_first_asterisk*
   - *7. take_till_first_asterisk*
   - *9. take_till_number*
   - *11. delete_what_follows_keeping,* where *list* ={" corp", " association", " participation", " co", " transaction", " inc", " inc.", " co.", " consulting", " consulting.", " ltd.", " ltd"}
   - *10. delete_what_follows,* where *list* ={" location", " total", " in", " our", " rescue", " loss", " has", " the", " notice", "\\#", " attached", " name", "

collision", " contracted", " damage", " contract", " to", " occupation", " facsimile", " file", " accident", " d\\/accident", " closing", " claim", " cat", " code", " open", " severe", " fatal", " fatality", " write", " we", " as", " process", " case", " send", " that", " most"}

- *12. delete_match*, where *list* ={"age.{1,1}dob", " dob", "injury", "injuries"}

3.2.2.4 INSUREDSYN

0. **pre-filtering**:
   - see the *document filtering* pseudocode below:

```
foreach document d having at least 5 sessions where the value in the
second session does not begin with an asterisk
        if value v begins with an asterisk then retain it
                                            else remove it
```

1. **filtering by synonyms**:
   - discard the *value* if *generated_from* ∈ {"ins. name"}
2. **filtering by regex**:
   - *2. delete_if_begin_with,* where *list* ={"\\$", "na ", "\\. ", "as ", "from ", "has ", "have ", "however ", "i ", "we ", "is ", "notwithstanding ", "once ", "went ", "\\*es ", "by ", "does ", "in ", "is ", "request ", "under ", "was ", "and \\wed ", "and will ", "contract ", "due ", "for ", "had ", "on ", "shall ", "which ", "would ", "or ", "information ", "of ", "who ", "page ", "section ", "usd ", "\\w\\?", "contact ", "owners ", "\\_\\.*", "should", "found ", "hit ", "any ", "that ", "\\d", "loss", "do ", "\\.{4,}", "at ", "claim ", "bill"}
   - *3. delete_if_end_with,* where *list* ={" na", " that", " the", " was", " this", " to \\w*", " at", " of", " he", " cmo\\.", " of", " will", " for", " which", " under", " fill", " so", " who", " further", " to", " because", " between", " arises", " under", " am", "\\-\\$", " every", " a", " \\w\\?", "could", "settle", "they"}
   - *4. delete_if_contain,* where *list* ={"\\-\\-\\-\\-", "township", " usd ", "\\|", "jurisdiction", " yes "}
3. **cleaning by regex**:
   - *6. clean_first_asterisk*
   - *7. take_till_first_asterisk*
   - *11. delete_what_follows_keeping,* where *list* ={" co.", " inc.", " corp.", " corp", " inc ", " corp ", " incorporated", " corporation", " company", " corporat", " co ", " county", " reinsurer", " construction"}
   - *10. delete_what_follows,* where *list* ={" participation", " master", " location", " direct", " invest*", " attached", " outstanding", " loss", " date", "\\*", "\\-tmp", " claim", " cna", " ui", " reinsurance", " contract ", " treaty", " producer", " claus", " status", " dear", " your", " x ", " retro", " number", " group", "\\- \\-", "\\w+\\d+\\w*", "\\d+\\w+\\d*\\w*", "\\d{5,}", " payment", " amount", "$\\d\\.*", " invoice", " number", "january ", "february ", "march

", "april ", " may ", "june ", "july ", "august ", "september ", "october ", "november ", "december ", " aka", " retro", " aww", " insured*", "\\#", " address", " carriers", " policies", " partne*", "class action", "employee", "check no", "item no", "idemnity", "shortfall", "\\;", " claim "};

- *12. delete_match,* where *list* ={"\\- \\.", "\\-\\.", "\\.+ ", "\\(.*\\)", "\\(.*"}

Moreover, as we observed that short values correspond usually to useless information, at the end of the filtering procedure we retain only those values having strictly more than 3 characters.

3.2.2.5 DATEOFLOSSSYN

Ad hoc procedure for date which is depicted in the pseudocode of Algorithm II.2.

```
foreach document d
  foreach record r
     replace all " - " with "-"
     for i=1,...,4
        vec[i] = apply data_match_i to the value v                    // regex [13-16]
        vec_clean[i] = date_cleanup(vec[i])                           // regex 5
     endfor (obtain a vector vec of four possible matchings)
    new_date = longest(vec)                 // retain only the longest string in vec
    std_date = to_standard_date(new_date)                             // regex 17
    output_date = date_cat(std_date)                                  // regex 18
    if trim(output_date) raises an error then output_date = ""
    endif
  endfor
endfor
```

Algorithm II.2. *Pseudocode of the date field filtering and cleaning processing. The trimming procedure may raise an error whenever output_date is not a string.*

3.2.2.6 CLAIMNUMBERSYN

The procedure for this key, using the below filters and cleaning procedures, is summarized in the pseudocode reported in Algorithm II.3.

1. **filtering by synonyms**:
   - discard the *value* if *generated_from* ∊ {"broker claim", "nbr", "clm"}
2. **filtering by regex**:
   - *4. delete_if_contain,* where *list* ={"\\$", "\\%", "(\\d{1}|\\d{2})\\/(\\d{1}|\\d{2})\\/(\\d{2}|\\d{4})", "(\\+\\d{1,2}){0,1}\\s{0,1}\\(\\d+\\)\\s{0,1}\\d*"}
3. **cleaning by regex**:

- *6. clean_first_asterisk*
- *7. take_till_first_asterisk*
- *10. delete_what_follows*, where *list* = {"paid loss", "paid boss", "reclaim", "invoice number", "your contract", "balance", "gpg", "amount", "direct loss", "issues.", "agency.", "fax", "facsimile no.", "tel", "invoice number", "our ref", "cedent", "direct loss", "sr business id", "amount paid", "index no", "incurred expense", " on ", "audit issues.", "reserve loss", "payment expense", "see attached", "previous", "dot", "reinsurer", "activity no", "phone", "suffix", "info", "reserve", "menu", "diary", "check numbers", "gross loss", "switchboard", "name", "invoice no.", "claim reference", "attachment", "limit", "my reference", "\\*s"}
- *12. delete_match*, where *list* ={"\\(.*\\)", "\\(.*", "\\[.*\\]", "\\[.*", "\\.\\.\\.\\.\\.\\.\\s{0,1}\\*"}
- *8. take_till_no_number*
- *19. claimnumber_match_1*
- *20. claimnumber_match_2*

```
filtering by synonyms
4. delete_if_contain
6. clean_first_asterisk
7. take_till_first_asterisk
10. delete_what_follows
12. delete_match
8. take_till_no_number
foreach value v
  vec[] = split v using space as delimiter
  vec1[] = apply 19. claimnumber_match_1 to vec[]
  vec2[] = apply 20. claimnumber_match_2 to vec[]
  vec_output[] = merge vec1[] with vec2[] by taking the longest elements in {vec1[i],
                                    vec2[i]} for i=1,...,sizeof(vec1[])
  apply 4. delete_if_contain and 12. delete_match  for each vec_output[]
  cat vec_output[] separating with a space the various items of the vector
endfor
```

Algorithm II.3. *Sequence of instruction to be executed for the claimnumber field filtering and cleaning processing.*

Final note

Note that the dimension of the resulting data structures, hence of the output files, is the same as that of the input files. In other words, no value is completely removed: filtering means substituting a value with the empty string “”.

3.2.3 Check points

Input: files *\check_point\post_process\output_name_key.csv*, where *name_key* ∈ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}.

Output: files *\check_point\post_process\output_name_key_post_filtering.csv*, where *name_key* ∈ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}.

3.3 Phase 3: final settlement

3.3.1 Description

The objective of this phase is to merge the whole information scattered in the output files produced in the previous phase, performing some further cleaning and, above all, grouping the values associated to the various keys within the session they refer to. In fact, on the one hand, each document may contain different sessions; on the other, according to the dynamic programming algorithm developed in part 1, a new session starts whenever a same key has been discovered in the document, regardless of the fact that this key may be a replica or, in any case, may refer to the same true document session (*practice*). Thus, as the discovered sessions may be a superset of the document practices, we propose here an heuristic to try to group all the meaningful keys in a restricted number of sessions. Details of such heuristic will be provided in a following implementation paragraph. Moreover, as this final phase is composed of various steps, we replicate here various *implementation – check point* sections, one for each step, for the sake of clearness.

3.3.2 Step 1: files merge

3.3.2.1 Implementation

We are left with the files *output_name_key_post_filtering.csv,* where *name_key* ∈ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}, containing the following columns: *doc_id*, *key_value*, *generated_from* and *position* (see Figure II.2 for a template reference). In order to merge those information in a new unique file, performing in the meanwhile some further cleaning and processing, the next steps must be followed:

1. for each file *output_name_key_post_filtering.csv*, select the columns *doc_id*, *key_value*, and *position,* thus discarding the field *generated_from.*
2. fill in all the *doc_id* fields: remember that only the first record contains information about the document identifier. The pseudocode is reported here below:

```
for each record r
    if the doc_id field is empty then copy therein the doc_id field of the record $r – 1$
    endif
endfor
```

3. retain for each value only the first 5 words, except for the DATEOFLOSSSYN key. It follows a prototypical pseudocode (see the Mathematica function `takeFive[]` and `filterFive[]`).

```
for each value v
    split v using space character as delimiter
    take at most the first 5 elements of the obtained vector discarding the rest
    join the remaining vectors using space character as word separator
endfor
```

for the DATEOFLOSSSYN key, discard any value concerning periods of times (i.e. having the form *date1 – date2 – ...*). For the sake of simplicity, we report here the related pseudocode (see the Mathematica function `takeFiveDate[]` and `filterFive[]`).

```
for each value v
    split v using “ - ” as delimiter
    if the resulting vector has more than 1 element then discard it endif
endfor
```

4. merge the cleaned files in a unique table, whose generic item (which we will refer to as *merge_item*) has the following form:

doc_id	val$_{11}$	pos$_{11}$	val$_{21}$	pos$_{21}$	val$_{31}$	pos$_{31}$	val$_{41}$	pos$_{41}$	val$_{51}$	pos$_{51}$
	val$_{12}$	pos$_{12}$	val$_{22}$	pos$_{22}$	val$_{32}$	pos$_{32}$	val$_{42}$	pos$_{42}$	val$_{52}$	pos$_{52}$
	...	...	...	...	...	...	...	...	...	...

where *doc_id* is the document identifier, val$_{ij}$ is the *j*-th value (i.e. the *j*-th occurrence found for a given key) associated to the *i*-th key, and pos$_{ij}$ is the corresponding position. See the Mathematica function `mergeRecord[]`.

Example II.1

Consider the *doc_id* 09010a1484de8248 having:

- ("metal stone construction", 2) as couple (value, position) associated to the key INSUREDSYN,
- ("",) – i.e. an empty entry – associated to the key CLAIMANTSYN,
- ("119603", 1) and ("cgl163168", 1) associated to the key CLAIMNUMBERSYN
- "04/02/2008", "3" to the key DATEOFLOSSSYN,
- ("",) to the key POLICYSYN.

The item of the merged table will be as follows:

09010a1484de8248	metal stone construction	2			119603	1	04/02/2008	3		
					cgl163168	1				

Or alternatively using Mathematica list notation:

```
{"09010a1484de8248", {{{"metal stone construction", "2"}, {"", ""},
{"119603", "1"}, {"04/02/2008", "3"}, {"", ""}}, {{"", ""}, {"", ""},
{"cgl163168", "1"}, {"", ""}, {"", ""}}}}
```

3.3.2.2 Check points

Note that, from now on, the checkpoint files will be saved in Mathematica "list" format. In case an inspection of such files is required, instead of opening them in a standard spreadsheet, it is recommended to use a text editor. For the sake of clarity, we use as file extension *.list* in case the adopted format is the aforementioned Mathematica "list".

Input: files *\check_point\post_process\output_name_key_post_filtering.csv*, where *name_key* ∊ {CLAIMNUMBERSYN, INSUREDSYN, CLAIMANTSYN, POLICYSYN, DATEOFLOSSSYN}.

Output: file *\check_point\post_process\output_merged.list*

3.3.3 Step 2: duplicate removal

3.3.3.1 Implementation

The term duplicate is somewhat misleading, as the aim of this step is to remove all similar values appearing in the same column of a given *merge_item*. Essentially, values associated to the same key and belonging to different document sessions may actually refer to the same document practice. To discover such situations, we adopt a simple heuristic, aiming at collapsing all values whose reciprocal similarity drops below a given threshold and retaining the smallest one (different from the empty string ""), as experimentally we observed that it is the one with the less redundant information). As measure of similarity we adopted a normalized form of the Damerau Levenshtein distance (see the Mathematica function `DamerauLevenshteinDistance[]` for its definition and a detail explanation of its functioning). In particular, given two strings *str1* and *str2*, their Normalized Damerau Levenshtein distance $d_{\mathrm{NDL}}(str1,str2)$ is defined as:

$$d_{\mathrm{NDL}}(str1,str2) = \texttt{DamerauLevenshteinDistance[}str1,str2\texttt{]}/\max(\mathrm{length}(str1),\mathrm{length}(str2))$$

The whole procedure is sketched in the pseudocode of Algorithm II.4.

```
for each document $d$
    for each key $k$
        for $i = 1,...,r$            // $r$ is the number of records (i.e. rows, values) in the merge_item
            for $j = i+1,...,r$
                if $d_{\mathrm{NDL}}(v[i],v[j]) < t$ then  // $v[i]$ is the value in the $i$-th row of merge_item, $t = 0.3$
                    // define $v^*$ as the shortest string between $v[i]$ and $v[j]$
                    let $v^* = \operatorname*{arg\,min}_{v \in \{v[i],v[j]\},\, v \neq ""} (\mathrm{length}(v[i]),\mathrm{length}(v[j]))$
                    $v[i] = v^*$
                    v[j] = “”
                endif
            endfor
        endfor
    endfor
    remove all empty records
endfor
```

Algorithm II.4. *Pseudocode of the duplicate removal step.*

3.3.3.2 Check points

Input: file *\check_point\post_process\output_merged.list*

Output: file *\check_point\post_process\output_merged_noduplicate.list*

3.3.4 Step 3: position renumbering

3.3.4.1 Implementation

The field *position* associated to each *value* describes, within each document session found by the dynamic programming algorithm in part 1, the within-session ordering of the keys, and consequently of the associated values, as discovered in the document. Namely, considering the key ordering (INSUREDSYN, CLAIMANTSYN, CLAIMNUMBERSYN, DATEOFLOSSSYN, POLICYSYN), if the first session of the considered document contains values associated to the keys DATEOFLOSSSYN, INSUREDSYN and CLAIMANTSYN (in this order), then the vector of position associated to each key is (2,3,“”,1,“”). Moreover, if the second session of the same document contains values associated to the keys CLAIMANTSYN and CLAIMNUMBERSYN (in this order), then the vector of position associated to each key is (“”,1,2,“”,“”). In other words, the position numbering restarts from the beginning (number 1) for each document session. The aim of the current step is to provide a univocal position numbering within the same document (namely, no two positions will be the same within the same document). In the above example, by adding *nr_key**(*sess_nr* – 1) to each position, where *nr_key* accounts for the number of keys (in our case *nr_key* = 5) and *sess_nr* is the session whereof we are currently listing the positions, we guarantee an unambiguous numbering; in fact, adding the constant 5*(2-1) to each element of (“”,1,2,“”,“”) we obtain (“”,6,7,“”,“”), whose elements are different from (2,3,“”,1,“”).

See the pseudocode in Algorithm II.5 for all the details, and Example II.2 for a typical output.

```
let nr_key be the number of keys                    // in this case nr_key = 5
for each document d
    for i = 1,...,r                 // r is the number of records (i.e. rows, values) in the merge_item
        let p(i) be the i-th vector containing the positions for all keys // i.e. (2,3,“”,1,“”)
        p(i) = p(i) + (i-1) * nr_keys
    endfor
endfor
```

Algorithm II.5. *Pseudocode of the position renumbering step.*

Example II.2

Given the *merge_item* of Example II.1, the output of Algorithm II.5 will be as follows:

09010a14 84de8248	metal stone construction	2			119603	1	04/02/2008	3		
					cgl163 168	6				

Or alternatively using Mathematica list notation:

```
{{{"metal stone construction", 2}, {"", {}}, {"119603", 1}, {"04/02/2008",  3},
{"", {}}}, {{"", {}}, {"", {}}, {"cgl163168",  6}, {"", {}}, {"", {}}}}}
```

where, for further processing, we have decided to replace any empty string “” referring to a position with an empty Mathematica list {}.

3.3.4.2 Check points

Input: file *\check_point\post_process\output_merged_noduplicate.list*

Output: file *\check_point\post_process\output_merged_renumbered.list*

3.3.5 Step 4: session condensing

3.3.5.1 Implementation

As already pointed out, the discovered sessions are (possibly) a superset of the true ones. We propose here a heuristic which has the benefit of reducing the number of sessions. In principle, given a *merge_item* as outputted in the previous step, focusing on the single columns (i.e. on the single keys) we may shift upwards all cells containing *true values* (i.e. values different from the empty string “” or from the empty list {}), so as to fill-in all the empty cells or those containing *spurious values* (i.e. values equal to the empty string “” or to the empty list {}). In such a way, we are free to delete, starting from the bottom of the considered *merge_item*, all the empty records. Unfortunately, using this simple strategy may lead to temporal inconsistencies, as some values discovered later in the document may now belong to a session *s1* preceding a session *s2* which in turn may contain values discovered earlier during the sequential reading of the document. Therefore, we should avoid all temporal inconsistencies by correcting such anomalies.
The proposed methodology is better introduced by a simple example, where we used as fictitious keys COLOR, SHAPE, and SIZE.

Example II.3

Given the following fictitious *merge_item*:

doc_id	yellow	1	rectangle	2	small	3
	blue	5	“”		big	4
	red	7	“”		“”	
	purple	11	square	10	“”	
	“”		rectangle	13	small	14
	“”		circle	17	medium	16

firstly we move up all the cells containing true values overwriting the content of those containing spurious values, obtaining as a result:

doc_id	yellow	1	rectangle	2	small	3
	blue	5	square	10	big	4
	red	7	rectangle	13	small	14
	purple	11	circle	17	medium	16

It is straightforward to note how some spurious associations between values belonging to different keys has arisen. For instance the second record contains a reference to a big blue square which cannot exists in the document as the square is surely placed hereinafter therein. In other words, we should avoid situations where a record contains positions greater than those occurring in subsequent records. If this occurs, we must shift down the value causing such anomaly (in this case (square, 10)), hence moving down all the subsequent values contained in the same column. We must repeat this repositioning until no rows contain positions greater than those included in the subsequent records. Coming back to the lead example, we obtain:

doc_id	yellow	1	rectangle	2	small	3
	blue	5			big	4
	red	7	square	10		
	purple	11	rectangle	13	small	14
			circle	17	medium	16

The whole procedure followed in the above example is highlighted in the pseudocode of Algorithm II.6 (mainly it is implemented in the Mathematica routine `moveUp[]`). In order to perform a definite cleaning of the spurious couples, i.e. those whose value corresponds to the empty string “” or whose position to the empty list {}, before executing Algorithm II.7 all spurious couples will be converted in (“”,0), i.e. the corresponding value will become the empty string and the position will be reset to 0.

```
let a couple (value, position) ($v$, $p$) be spurious when $v$ = {} or $v$ = “”
for each document $d$
    for each key $k$
        for each record $r$
            let $v[k,r]$ be the $r$-th value associated to the key $k$ in the merge_item
            let $p[k,r]$ be the $r$-th position associated to the key $k$ in the merge_item
            if $v[k,r]$ is spurious then move up all cells $v[k,r']$ and $p[k,r']$ with $r' > r$
        endfor
    endfor
    for each record $r$
        for each key $k$
            if exists at least a $k'$ such that $p[k',r+1] > p[k,r]$ then
                move down all cells $v[k,r']$ and $p[k,r']$ with $r' \geq r$
            endif
        endfor
    endfor
endfor
```

Algorithm II.6. *Pseudocode of the session condensing step. Note that the terms* move up *and* move down *means moving a cell in the record (row) above or below, respectively.*

A typical *merge_item* after the execution of Algorithm II.7 is reported in the following Example II.4.

Example II.4

The execution of Algorithm II.7 will produce as output a *merge_item* like the one shown below in tabular form:

09010a1484de8555	stout roofing of california inc construction	1	vcc newark co	2	“”	0	“”	0	lha1 23000	4
	“”	0	“”	0	“”	0	01/10/2003	11	lha1230	6
	“”	0	“”	0	“”	0	“”	0	lha123909 -	12

which, using Mathematica list notation, becomes:

{"09010a1484de8555",{{{"stout roofing of california inc", 1}, {"vcc newark co", 2}, {"", 0}, {"", 0}, {"lha1 23000", 4}}, {{"", 0}, {"", 0}, {"", 0}, {"01/10/2003", 11}, {"lha1230", 6}}, {{"", 0}, {"", 0}, {"", 0}, {"", 0}, {"lha123909 -", 12}}}}

3.3.5.2 Check points

Input: file *\check_point\post_process\output_merged_renumbered.list*

Output: file *\check_point\post_process\output_merged_condensed.list*

3.3.6 Step 5: final file preparation

3.3.6.1 Implementation

In this last step, a very simple final processing is performed to produce a self-contained well-formed *csv* file. In detail:

- each record of a *merge_item* should contain information about the *doc_id* and the session it refers to;
- all positions should be removed: in other words only the value information must be retained for each key;
- whenever a value is empty, if there exists a previous session, the value reported therein (referring to the same key) must be reported.

Example II.5

With reference to the *merge_item* shown in Example II.4, after the three above steps we obtain the following tabular form:

09010a14 84de8555	1	stout roofing of california inc construction	vcc newark co	“”	“”	lha1 23000
09010a14 84de8555	2	stout roofing of california inc construction	vcc newark co	“”	01/10/2003	lha1230
09010a14 84de8555	3	stout roofing of california inc construction	vcc newark co	“”	01/10/2003	lha123909 -

whose header would reads:

doc_id	*sess_nr*	**INSURED SYN**	**CLAIMANT SYN**	**CLAIMNUMBER SYN**	**DATEOFLOSS SYN**	**POLICY SYN**

Again, in Mathematica list notation, it reads:

```
{{"09010a1484de8555", 1, "stout roofing of california inc", "vcc newark co", "", "", "lha1 23000"}, {"09010a1484de8555", 2, "stout roofing of california inc", "vcc newark co", "", "01/10/2003", "lha1230"}, {"09010a1484de8555", 3, "stout roofing of california inc", "vcc newark co", "", "01/10/2003", "lha123909 -"}}
```

For the sake of completeness, we report in the following Algorithm II.7 the pseudocode of the three steps aimed at preparing the final file.

```
for each document d
    remove all the columns containing any position information
    add a column containing the session nr. sess_nr (after the first column doc_id)
    for each record r
        insert a sequential number in the sess_nr column, starting from 1
        for each key k
            let v[k,r] be the r-th value associated to the key k in the merge_item
            if v[k,r] contains the empty string “” and r > 1 then v[k,r] = v[k,r – 1] endif
        endfor
    endfor
endfor
```

Algorithm II.7. *Pseudocode of the final file preparation step.*

3.3.6.2 Check points

Input: file *\check_point\post_process\output_merged_condensed.list*

Output: file *\results\output_complete.csv*