Clustering of handwritten words based on structural features extraction

Lorenzo Cioni, Francesco Santoni

lore.cioni@gmail.com, fsanto92@hotmail.it

19 January 2015

Abstract

In this report we will discuss a method of extracting primitive features from handwritten characters to generate clusters of similar words, in our case the names of American States found in the 1930' census.

Contents

1 Introduction

Segmentation and clustering of large amounts of data is one of the main research fields of modern artificial intelligence.

The basis of our work is the collection of U.S. Census data in the year 1930 and fits into the process of digitalization of handwritten documents that characterizes our time. In this case we have scans of the census register and the main goal is to divide enrollees by State.

Document segmentation and extraction of the word corresponding to the State was developed previously by two of our colleagues in the course of *Technology of Databases* and is the basis from which our work started. The problem on which we have worked is the extraction of features from handwritten characters with the goal of creating clusters of similar words in order to facilitate the recognition by a human agent.

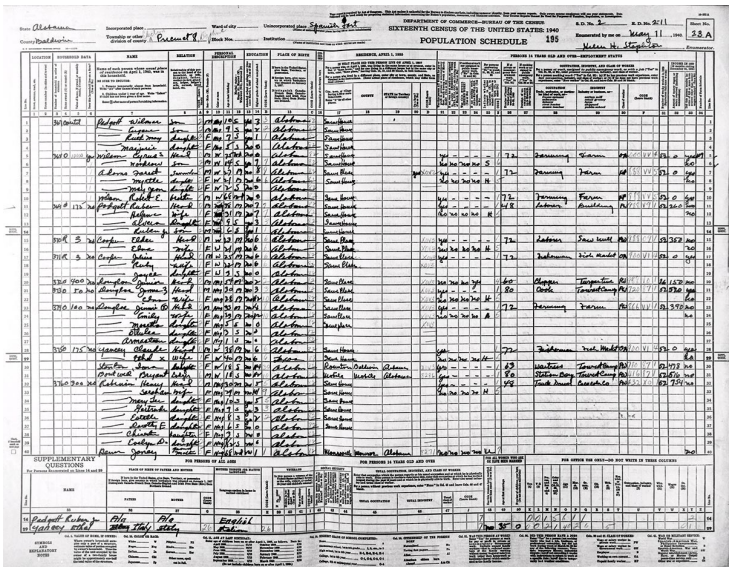


Figure 1: An example of a table containing census data

2 The pre-existing project

The census page that contains the data in digital form holds a wealth of information about each person surveyed: name, gender, membership status and some secondary features such as work or the breed.

All these data are housed in a moulded grid composed of numbered rows and columns. In addition to the citizens data, each archive also contains additional information related to the censor or data relating to samples of the population.

The project's objective was finding a particular grid area, corresponding to the area containing the registered State of each person, from which extract the handwritten text.

Following the words localization, the existing work proceeds by grouping visually similar elements to form a collection of homogeneous samples, in hope that each set will then contain cut outs of the same State.

The goal of the work is not to group all the words corresponding to the same state in a single cluster but rather to generate clusters that contain words that belong to a single state, the major requirement of the work is thus precision in the retrieval steps.

2.1 Process steps

We can summarize the project in three basic steps:

1. Localization of the text area containing the words of interest
2. Row segmentation
3. Post-processing and clustering

The last step, corresponding to the clustering of images, is the point of interest of our paper: while the past work was focused on the localization and retrieval not much thought was put on the features to be extracted preferring simplicity and efficiency to effectiveness. It is this very point that gives our paper a reason d'etre: starting from the preceding, already implemented, two steps, we proceed by providing a rewritten and improved clustering method through the use of new features, these steps will be discussed later in the section 3.

2.1.1 Column localization

In this first phase the aim is to identify the region of the grid within which the State words are located.

First of all the black border of the document is removed, an artefact resulting from the physical scanning. Then, through a vertical projection of the pixels and the creation of an histogram, the grid columns are located and, knowing the correct column's offset regarding the beginning of the document, just the one concerned is extracted.

2.1.2 Row segmentation

After the extraction of the concerned column from the document, we proceed with row segmentation.

Similar to the previous step, but using an horizontal projection this time, we are able to determine with some accuracy the rows of the grid. In this case, however, it's necessary to centre the word in the extracted image: this is done by correcting the height of each row by the analysis of black spikes on the created histogram.

At the end of this phase, ideally each word is contained in an individual image.

2.1.3 Post-processing and clustering

In the post-processing phase the individual images extracted are reworked in order to remove any vertical or horizontal residual lines left from an imperfect previous cut.

Delete row/column strokes can be about as bring back the black pixels related to the residual value of pure white (255 as grayscale).

This allows us to extract the most significant features in the next steps of processing.

Each image is divided in windows 1 pixel wide, on these windows the features are then extracted. The idea is not placing excessive burden on the operating system, the features are thus relatively simplistic: for each window the value of the height of the first and last black pixels are collected, together with the number of transitions of the pixels from black to white and vice versa. The distance between images is then found through the Euclidean Distance where the considered axis are the number of transitions and the stroke height calculated through a simple subtraction of the first and last black pixels height. The distance matrix is then fed to the Affinity Propagation algorithm to create the desired clusters.

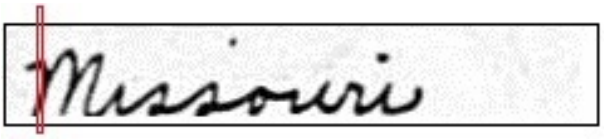


Figure 2: An image extracted from document, the red box delineates the 1 pixel wide window.

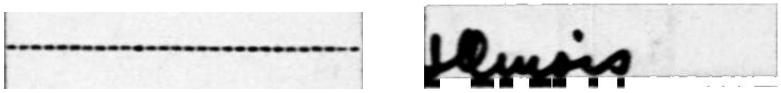
As you can see in Figure 2, the word is centered and the extra line have been removed (there is a line set to white). For each picture we proceed with feature extraction for handwritten characters and clustering.

2.2 Problems

The localization of words and their segmentation has inherent strong difficulties.

A first problem is represented by the document skew. In this case, the skew may be due to a not perfectly horizontal scan of the original document. The presence of skew reduces the capability in searching for rows and columns of the grid, thus making the segmentation impossible with the given implementation. In some cases, for example, the first column is interpreted incorrectly causing the extraction of the wrong column in the first stage of the process thus dooming that particular word recognition.

The main problems of the second phase are mainly related to the identification of extraneous lines. Because of the large number of dashed lines present it isn't always possible to *clean* the images and this can cause errors in the clustering phase.



(a) An image with a dashed line (b) The line intersects the word

Figure 3: Segmentation and post-processing issues

In other cases the word intersects directly gridlines. Because of this we may experience a loss of information about words if the line is removed using the method described above.

3 Features extraction

In this work we want to improve upon the past project by providing an alternate method to calculate the distances, the focus isn't any more simplicity but even at a major cost

we want to find characteristics more deeply related to the written words nature in hope they may be better suited for the recognition of similar written words.

The idea is thus to find *primitive* features, resembling the possible types of strokes used to write a word, to characterize the sample from which they are extracted in order to perform clustering on their set.

3.1 Structural features

The features are identified by the study of a sub-portion, or window, of the images correspondent to the entries of the "State" field of the census document on which the procedure is applied.

The features we search are intrinsically characteristics of the stroke, they are thus linked to a particular position in the image in which that particular stroke is found, as such they are located through the study of the area, the window, with which they are implicitly associated, this poses a problem: there is the possibility that an area may be characterized by multiple features, in this case there is no clear order of which of the multiple features comes first. Due to the absence of such *native* order the string that defines a window is maintained consistent with the other strings trough the convention of generating the string with the features' identifiers always taking the same order, if present.

The chosen way to resolve the issue however presents the problem of making sliding windows inapplicable: this means that a sample must necessarily be cut in separate windows, which are allowed to overlap, with the corresponding effect that due to the random cut some features like *loops* may not be recognised in an instance and recognised in another.

Each sample is associated to a characterizing string of characters made from the identifiers of the features recognized in the sample. The features that convey more information about a word, for example loops and dots, are given longer strings so that their presence may be better recognised.

The string is constructed modularly by appending in order the strings associated to each window the sample is divided in.

After having recognised the presence of a feature in a given window the corresponding identifier is added to the end of the string correspondent to the window; the order in which the features are searched and thus added to the queue is decided a priori and maintained consistent during the construction of the strings.

Windows Currently we have chosen to cut the samples in windows of 40 pixels each, with the exception of the last part of the sample that, deemed irrelevant to the end of characterization due to containing almost always white space, is simply ignored.

These windows are spaced only 7 pixels from the preceding and succeeding one, with the intent of creating overlap and lengthening the string that characterizes the samples. (Both the dimension of the window and the spacing between them was decided through tests to obtain the optimal efficacy of the program)

While the simplest way to create the windows in the code would be simply cutting the original image in pieces these steps require memory and time. We have thus preferred to increase the complexity of the functions that search for the features, to whom we pass as a variable the original images with information about the *offset* at which to start the search and the *width* of the window. The width of the window passed has actually a non

banal meaning due to the fact that different features are linked to areas of the image with varying dimensions: for example it is not efficient to search for an horizontal line and a vertical line utilizing windows with the same width due to the fact that horizontal lines realistically will require windows with great width but will not have requisites on the height.

In particular, the features with less space requirements are searched in both 20 pixels halves of the windows and their strings are appended to the image string the whole 40 pixels window is then searched for the space-hungry features and their strings are added to the "counter".

The height of the windows used is never considered as a parameter since for all the features searching functions the height is customarily the whole height of the sample.

3.1.1 Whitespace

The first feature that is searched in the windows is the white space. This way if a window is identified as blank there won't be a need to proceed with the search of the other features, saving time.

Figure 4: A whitespace

A window is identified as blank if the average pixel values is below a pre-set threshold. The string associated with the Whitespace is " ".

3.1.2 Loop

The feature that represents a loop requires the search of the window with greater size. The localization of a loop starts by searching for a black pixel which is then tested appropriately. The idea behind is that a point on the border of a loop is such that by looking in a decided direction (in this case the y axis) we meet two pixel value transitions first from black to white and then from white to black. Between the two transitions for a loop to be recognised there must be a minimum number of white pixel, number that is decided discretionally in the code.

Figure 5: A loop

To recognise a loop the above condition must also be verified in the other cardinal axis. To do so, starting from the center of the supposed loop (the median value of the segment described above) we check that moving either right or left we find a transition from white to black after a suitable number of white pixels.

The implemented method has numerous problems related to the difficulty of finding the best white spaces threshold: an excessively little minimum number of white pixels makes us recognise as loops white noise that happens while scanning images, a threshold too high forces us to discard some small but real loops. The method also does not take into account the thickness of the stroke, rendering the overall recognition harder, with appropriate threshold values tough we can achieve good results.

The string associated with the Loop feature is "LL".

3.1.3 Dot

To search for Dots within the segment we first proceed locating the *connected components* inside of it. The individual components are extracted and inserted into a *box* of which we know the size and the relative position to the segment.

Figure 6: A dot

At this point we search for those boxes with dimensions between two pre-set values (minimum and maximum radius), those that meet this condition are boxes that contain most likely points.

Using the connected components we can thus retrieve dots in a simple way.

The string associated with the Dot feature is "....." It is particularly long because the dot helps distinguish the words that contain the "i" letter and is easily recognised with little error.

3.1.4 Diagonal line

The feature representing a diagonal line, both upward and downward facing, is extracted through a simple exhaustive scansion of the window for lines of connected black pixels that have an incline in a range of values and are sufficiently long.

At the moment the function doesn't account for the width of the lines found, thus diagonal features can be recognised in a shapeless blob of black pixel that is sufficiently big.

Figure 7: A diagonal line

There's a distinction for diagonal features that appear in the lower and upper bottom of the window. At most in a window the function identifies a couple of upward and a couple of downward facing lines (lower and upper parts), once one has been found it stops searching for the same type. The lines that are found are further categorised on their length if medium or long.

Many strings are associated with the Diagonal feature, they will be properly listed in the Table 1. In particular the string associated with the "long" version of a feature is made from the "medium" version of the string at which is attached an other character: this allows to distinguish features based on their length while at the same time maintaining similarity with a shorter version of themselves to account for variation in the stroke used to write the same word.

3.1.5 Cross

The function, having found at most a lower and upper case for upward and downward diagonal lines, proceeds to confront the edge points of these lines: if they possibly intersects it extracts a *crossing* feature with distinction if the crossing happens in the lower or upper part of the window.

The problems with this sub-feature are that intersections of bottom and upper lines of the same type (e.g. both upward facing) with different inclines are not recognised, in the same way there's no consideration for intersections made from lines that are not the "primary" bottom and upper diagonals: if in a single windows are present more lower upwards diagonals and one of them other than the first intersects the recognised downward diagonal, such a crossing is ignored.

The crossing feature may also not necessarily be a complete intersection, in fact for recognition there just needs to be a merging of a downward and upward line.

The string associated depending if the cross is found in the upper or lower half is "X" and "x" respectively.

3.1.6 Horizontal and Vertical line

Both horizontal and vertical lines' features are extracted through a simple scan of the window.

The horizontal lines requires a double-window for their implicit characteristic. The function identifies a line if it finds a connected row or column of black pixels that has sufficient length, in particular it distinguishes the lines found in normal or long through an ulterior threshold.

Once a fitting line has been found the function keeps searching for more, continuing the scansion of the window after having moved a certain distance from the last line found in order not to confuse a particularly thick stroke as different separate lines.

There still persists the problem that the stroke width is not fully considered so a big blob of black pixels is seen as a series of horizontal and vertical lines, at the same time such an occurrence is rare so the presence of many horizontal and vertical lines ends up distinguishing the word in itself.

Feature	**Associated string**
Whitespace	" "
Loop	"LL"
Dot	"....."
Upward facing diagonal lower half, medium	"s"
Upward facing diagonal lower half, long	"bs"
Upward facing diagonal upper half, medium	"S"
Upward facing diagonal upper half, long	"BS"
Downward facing diagonal lower half, medium	"u"
Downward facing diagonal lower half, long	"vu"
Downward facing diagonal upper half, medium	"U"
Downward facing diagonal upper half, long	"VU"
Cross upper	"X"
Cross lower	"x"
Horizontal line lower half, medium	"-"
Horizontal line lower half, long	"h-"
Horizontal line upper half, medium	"H-"
Horizontal line upper half, long	"Hh-"
Vertical line lower half, medium	"ii"
Vertical line lower half, long	"IIii"
Vertical line upper half, medium	"Vii"
Vertical line upper half, long	"VIIii"

Table 1: Recap of the string associated to the features

3.2 Dimensional features

While the structural feature already proposed help in categorizing and recognising words to cluster they aren't completely able to distinguish a word from similar others, we have

thus searched for ways to improve the accuracy of the work: while many optimizations have been added to the structural features sought themselves we have found that adding another layer of features of a different kind, in this case *dimensional*, helps tremendously. Moreover the complexity added is extremely limited: as can be seen later on in the Table 4 the »><««CONTINUA«» In some cases the structure feature extraction leads to some problems of recognition or it may not be sufficient to properly assess the similarity between two words.

In order to improve word clustering we combine structural features with *dimensional features* for each word. These dimensional features are exactly the ones previously utilized in the pre-existing work: height of the stroke, calculated through the difference between highest and lowest black pixel, and number of transition of the pixels' value from black to white and vice versa.

The explanation can be found in section 2.1.3

3.3 Features extraction example

We present now an example of extraction of features from a word. We initially create, as described above, a sliding window that scrolls along the image by a fixed step. Than, for each section, features are extracted and a substring is generated.

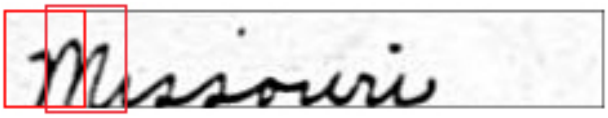


Figure 8: Feature extraction from *Missouri* word

(a) (b) (c) (d) (e) (f) (g) (h) (i) (j) (k) (l) (m)

Figure 9: Sliding window segmentation

(a) Some **diagonal lines** (ascending (s) and descending (u)), at the top (S) and at the bottom (s) of the image. Generated string: "*sSUusSUSu*".

(b) As the previous image and two more diagonal lines representing the "i". An **horizonal line** (H). Generated string: "*sSUusSUSuHsu*".

(c) Some diagonal lines and, at the end of the window, an horizontal line. Generated string: "*ssSusSsH*".

(d) Two consecutive similar character represented by diagonal lines and **vertical lines** in the middle. Generated string: "*sVHsV*".

(e) In that window we can find a **dot** (.). Generated string: "*ssVHs.*"

(f) The same previous dot and a little **loop** (L) at the bottom. Generated string: "*ssVs.LssH*".

(g) The same loop as previous, an horizontal line a vertical line and two diagonal. Generated string: "*LVHssu*".

(h) Generated string: "*ssVHus*".

(i) The other dot here. Generated string "*ssus.H*".

(j) An i. Generated string: "*ssuu.*".

(k) Only a diagonal line. Generated string: "*s*".

(l) Empty window. Generated string: " ".

(m) Empty window. Generated string: " ".

Once those strings are generated they are combined together to generate the word *structure string.*

4 Evaluating distances

4.1 Longest Common Subsequence

In order to compare the generated strings one another we must define a distance on the samples. The distance used in our work is based on the *Longest Common Subsequence* (**LCS**) algorithm.

The LCS algorithm has the aim of extracting from a set of sequences (in this case only two) the longest common subsequence, that is a sequence that is obtainable from both the starting sequences by deleting some elements without changing the order of the remaining ones.

LCS is a particular case of the *Edit Distance* algorithm where the only allowed operations are insertion and deletion. The distance associated with LCS in our current work is the number of insertion and deletions that must be applied to obtain the longest subsequence, in accordance with the Edit distance where the distance is calculated with the number of operations needed to morph a string in the other (in Edit Distance it's possible moreover to confer customizable costs to the substitutions).

While the LCS algorithm may require high costs when applied concurrently to high numbers of sequences, in our case there exists an easy and light implementation that exploits *dynamic programming*, in this type of implementation the cost ends up being $O(n * m)$ where n and m are the length of the compared strings.

$$LCS(X_i, Y_j) = \begin{cases} 0 & if\ i = 0\ or\ j = 0 \\ LCS(X_{i-1}, Y_{j-1}) \cup x_i & if\ x_i = y_j \\ longest(LCS(X_i, Y_{j-1}), LCS(X_{i-1}, Y_j)) & if\ x_i \neq y_j \end{cases}$$

Obtained the Longest Common Subsequence between two strings the distance between them is thus:

$$x.length() + y.length() - 2 * LcsLength$$

where x and y are the strings and the $length()$ function returns the length of a string.

4.2 Euclidean distance

Longest Common Subsequence is only used with strings. In order to improve the correctness of words distance we combine LCS distance with euclidean distance which uses dimensional features insted of structure features.

The eucliden distance is evaluated with

$$d = \sqrt{((a_b - a_t)(b_b - b_t))^2 - (a_c - b_c)^2}$$

with

- a_b representing the end of the stroke a
- a_t representing the begin of the stroke a
- b_b representing the end of the stroke b
- b_t representing the begin of the stroke b
- a_c representing the number of transition (normalized) of a
- b_c representing the number of transition (normalized) of b

At this point we evaluate the L1 normalization for each segment we proceed evaluatin distances from different features with the previous way.

5 Clustering

The clustering phase consists in the categorization of the various segments in homogeneous groups, so that, at the end of the process, each cluster contains words corresponding to the same State.

At this stage the main goal is the extraction of good features representative of the images to be used for comparison. The aim is to construct a similarity matrix between different segments so that they can be used as input to the clustering algorithm. To do so we generate a measure of distance between couples of samples through the use of the *Longest Common Subsequence* algorithm in which the handled strings, representative of the corresponding samples, are constructed through the appending of conventional identifiers associated with the features found in the samples in a consistent order.

Unable to establish a priori the optimal number of desired clusters makes the use of the *Affinity Propagation* algorithm a necessity.

5.1 Affinity Propagation

Affinity Propagation is a clustering algorithm that identifies a set of *exemplars* that represents the dataset[1]. The input of Affinity Propagation is the pair-wise similarities between each pair of data points, $s[i,j] \forall i,j = 1,\ldots,n^2$. Any type of similarities is acceptable thus Affinity Propagation is widely applicable.

Given similarity matrix $s[i,j]$, Affinity Propagation attempts to find the exemplars that maximize the net similarity, i.e. the overall sum of similarities between all exemplars and their member data points. The process of Affinity Propagation can be viewed as a message passing process with two kinds of messages exchanged among data points: *responsibility* and *availability*.

Responsibility, $r[i,j]$, is a message from data point i to j that reflects the accumulated evidence for how well-suited data point j is to serve as the exemplar for data point i.

Availability, $a[i,j]$, is a message from data point j to i that reflects the accumulated evidence for how appropriate it would be for data point i to choose data point j as its

[1] Brendan J. Frey, Delbert Dueck, *Clustering by Passing Messages Between Data Points*, http://www.sciencemag.org/, 2007

[2] $s[i,j]$ for each data point is called preference and impacts the number of clusters.

exemplar. All responsibilities and availabilities are set to 0 initially, and their values are iteratively updated as follows to compute convergence values:

$$r[i,j] = (1-\lambda)\rho[i,j] + \lambda r[i,j]$$
$$a[i,j] = (1-\lambda)\alpha[i,j] + \lambda a[i,j]$$

where λ is a damping factor introduced to avoid numerical oscillations, and $\rho[i,j]$ and $\alpha[i,j]$ are, we call, *propagating responsibility* and *propagating availability*, respectively.

$\rho[i,j]$ and $\alpha[i,j]$ are computed by the following equations:

$$\rho[i,j] = \begin{cases} s[i,j]\max_{k=j} a[i,k] + s[i,k] & i \neq j \\ s[i,j]\max_{k=j} s[i,k] & i = j \end{cases}$$

$$\alpha[i,j] = \begin{cases} min0, r[j,j] + \sum_{k\neq i,j} \max 0, r[k,j] & i \neq j \\ \sum_{k\neq j} \max 0, r[k,j] & i = j \end{cases}$$

That is, messages between data points are computed from the corresponding propagating messages. The exemplar of data point i is finally defined as:

$$arg\max r[i,j] + a[i,j] : \forall\, j = 1, 2, \ldots, n$$

As described above, the original algorithm requires $O(n^2t)$ time to update massages, where n and t are the number of data points and the number of iterations, respectively. This incurs excessive CPU time, especially when the number of data points is large[3]. The Figure 10 shows how affinity propagation works[4].

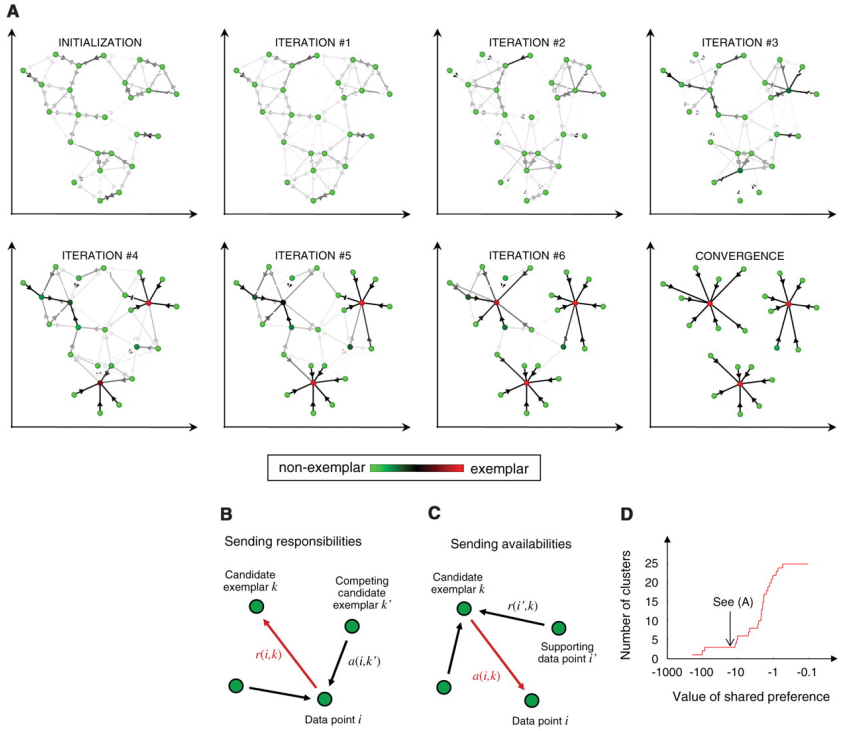


Figure 10: How affinity propagation works

[3]Yasuhiro Fujiwara, Go Irie, Tomoe Kitahara, *Fast Algorithm for Affinity Propagation*, 2009

[4]Brendan J. Frey, Delbert Dueck, *op. cit.*, Figure 1, p. 974

The clustering process described is then applied to the array of similarity calculated previously on Features extracted from each crop. Once you have run the calculation of clusters, segments are organized into individual folders to provide a Visual result of the proceedings just completed. Each group is identified by a segment that represents the centroid of the cluster, which is the element to which all others in the group are closer.

6 Results

Let's explore the test phase and see the results in detail. Initially we ran debug tests on our personal pc by porter quickly modify some code.

All tests will be shown below were performed on a single machine that has enabled us to work with much larger data in less time. The machine used is composed of two Xeon processors for a total of 16 cores @ 2.80Ghz and 48 Gb of RAM.

The tests were performed on a growing number of scans, each containing a maximum of 50 lines from which we extract the word that represents the State. For each number of different scans have been tried the three possible distances calculation: only LCS, only L1 (Euclidean distance) and LCS and L1 combined together.

The data collected for each test are as follows:

- **Number of scans**: the number of scans that was carried out the test.
- **Estimated words**: the number of words estimated on the basis of the number of scans.
- **Extracted words**: the number of words actually mined and processed, as well as a percentage.
- **Number of clusters**: the number of clusters created in process.
- **Features extraction time**: the time, in seconds, required for the extraction of features from the words.
- **Evaluating distances time**: the time, in seconds, required to generate the similarity matrix with the distances between all the words extracted.
- **Clustering time**: the time, in seconds, required for the creation of clusters.
- **Running time**: the total execution time, in seconds.
- **Correct clusters**: the number of clusters with maximum precision, therefore containing only words equal to each other.
- **Correct elements**: the number of words within clusters corrected. Even as a percentage.
- **Precision**: the accuracy of the result, the average accuracy of individual clusters weighted with the number of words.

In the next table we're going to show the main results for the tests.

[5]During this test, the machine used was concurrently executing other tasks creating a bottleneck in the memory used.

	Estimated words	Extracted words	Number of clusters	Running time (s)	Precision (%)
16 scans (L1)	800	550	55	15.52	58.00
16 scans (LCS)	800	550	71	44.39	60.00
16 scans (LCS and L1)	800	550	66	47.41	62.91
32 scans (L1)	1600	800	67	38.58	52.87
32 scans (LCS)	1600	800	92	94.54	55.50
32 scans (LCS and L1)	1600	800	86	112.90	56.25
75 scans (L1)	3750	2350	173	221.03	63.65
75 scans (LCS)	3750	2350	189	1169.72	67.49
75 scans (LCS and L1)	3750	2350	199	1203.28	71.16
130 scans (L1)	6500	5050	425	5444.43	71.12
130 scans (LCS)	6500	5050	441	5629.71	74.41
130 scans (LCS and L1)	6500	5050	463	6729.62	77.94
500 scans (L1)	0	0	0	0	0
500 scans (LCS)	0	0	0	0	0
500 scans (LCS and L1)	25000	18146	3941	138309.85[5]	82.14

Table 2: Main results

As we can see in Table 2 the number of extracted words is much less than the estimated number of words. This is mainly due to the fact that not all census tables are filled in full: in some cases there are only a few lines or has not been filled in the status column. In other cases, the error is due to a difficulty in removing the word from the scan due to a wrong interpretation of the rows.

As we can see in Figure 11 the accuracy of the cluster grows with the amount of words extracted. This phenomenon is due to the fact that Affinity Propagation works best with a large number of available data: the greater the number of words, the greater the chances of finding words similar between them, and then combine them within a single cluster.

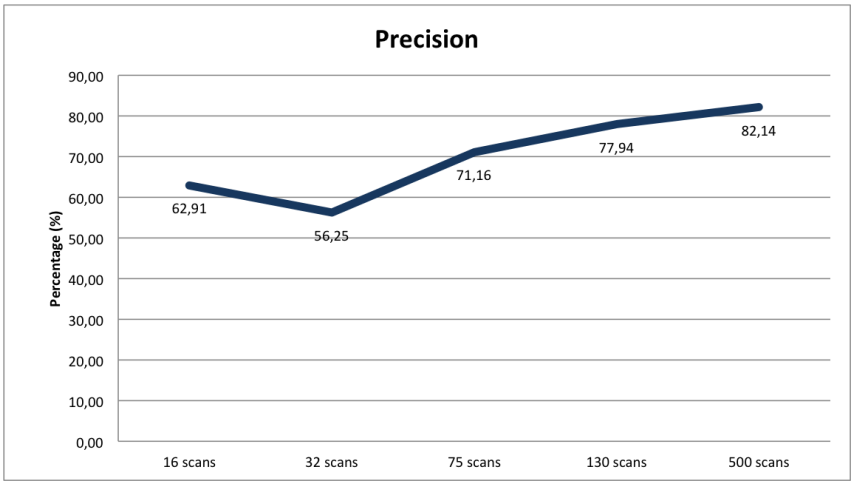


Figure 11: Clustering precision based on number of scans

Now we're going to focus our attention to time. The running time is divided mainly into three distinct phases: time for the extraction of features, time for the creation of similarity matrix (calculation of distances) and clustering.

As we can see in Table 4 the computational time is mainly due to the construction

	Features extraction time (s)	Evaluating distances time (s)	Clustering time (s)	Total (s)
16 scans (L1)	6.81	5.02	3.69	15.52
16 scans (LCS)	6.84	34.40	3.15	44.39
16 scans (LCS and L1)	6.84	37.32	2.85	47.41
32 scans (L1)	11.54	7.22	19.82	38.58
32 scans (LCS)	11.27	74.72	8.55	94.54
32 scans (LCS and L1)	11.34	89.96	11.60	112.90
75 scans (L1)	33.65	69.00	118.38	221.03
75 scans (LCS)	35.13	1002.63	131.96	1169.72
75 scans (LCS and L1)	34.70	1070.92	97.66	1203.28
130 scans (L1)	70.80	313.70	5059.93	5444.43
130 scans (LCS)	68.17	4353.85	1207.69	5629.71
130 scans (LCS and L1)	73.10	5608.82	1047.70	6729.62
500 scans (L1)	0	0	0	0
500 scans (LCS)	0	0	0	0
500 scans (LCS and L1)	403.41	61424.57	76481.87	138309.85

Table 3: Running time

phase of similarity matrix, so the calculation of distances. This occurs especially in calculating LCS distance due to the fact that structural strings of words are very long and the cost of the algorithm is $O(nm)$, with n and m the lengths of the two strings.

Structural long strings characterize better the word (and therefore make more accurate clustering), but involve a longer calculation time.

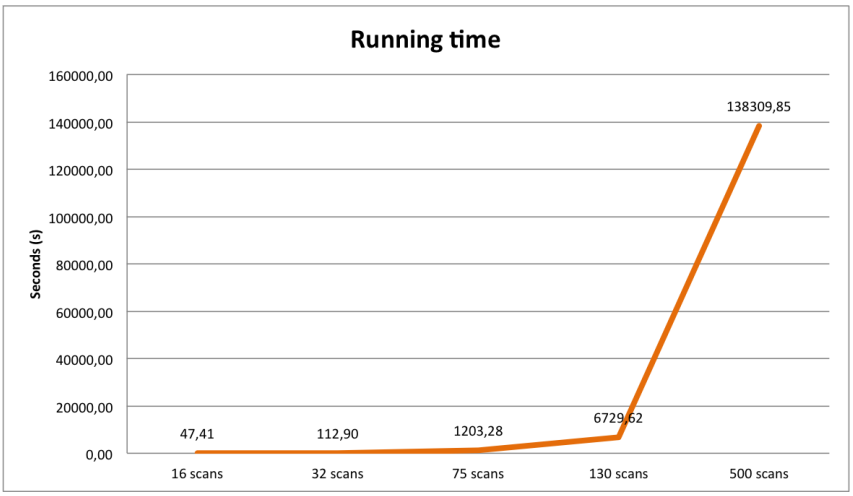


Figure 12: Running time on number of scans

A parameter to evaluate the goodness of the newly formed clusters can be the number of clusters that have a maximum precision, or which contain all similar elements between them and properly classified. This parameter is indicative as it can still exist cases of *false negatives* (words properly classified but labelled incorrectly) that then cut out the entire cluster.

In the following table are collected the number of clusters that respect the above described property and the number of items that they contain.

	Correct clusters	Correct words	Percentage of words (%)	Average words for cluster
16 scans (L1)	12	18	3.27	1.50
16 scans (LCS)	33	44	8.00	1.33
16 scans (LCS and L1)	25	45	8.18	1.80
32 scans (L1)	11	11	1.38	1.00
32 scans (LCS)	39	48	6.00	1.23
32 scans (LCS and L1)	32	43	5.38	1.34
75 scans (L1)	39	172	7.32	4.41
75 scans (LCS)	78	265	11.28	3.40
75 scans (LCS and L1)	77	316	13.45	4.10
130 scans (L1)	138	651	12.89	4.72
130 scans (LCS)	208	713	14.12	3.43
130 scans (LCS and L1)	215	860	17.03	4.00
500 scans (L1)	0	0	0	0
500 scans (LCS)	0	0	0	0
500 scans (LCS and L1)	3020	7616	41.97	2.52

Table 4: Correct clusters

7 Compiling and running notes

This software was developed entirely in C++, and uses an open source library of image processing and analysis, *Leptonica*[6], version 1.70.

Leptonica provides many functions for manipulating images pixel by pixel using a high-level approach. Thanks to this library for example, you can draw up a diagram of projections, crop images, or find the connected components in a portion of the image.

To compile the code, once included the library described above, it is necessary, in the case of version of *GCC/G++* less than 4.7, compile with version 11 of C++ that introduces support for threads.

To run this programm use the following parameters:

- **-d** to specify the images directory.
- **-t** to specify the number of threads. Default is 2.
- **--lcs** to use only LCS distance for building similarity matrix.
- **--l1** to use only Euclidean distance for building similarity matrix.

8 Conclusion

[6]Leptonica,a pedagogically-oriented open source site containing software that is broadly useful for image processing and image analysis applications -http://www.leptonica.org/