String averages and self-organizing maps for strings

Igor Fischer, Andreas Zell
Universität Tübingen
Wilhelm-Schickard-Institut für Informatik
Köstlinstr. 6, 72074 Tübingen, Germany
{fischer, zell}@informatik.uni-tuebingen.de

Abstract—

In a recent paper, T. Kohonen and P. Somervuo have shown that self-organizing maps (SOMs) are not restricted to numerical data. They can also be defined for symbol strings, provided that one defines an average function for strings and that the adaptation process is performed off-line (batch). In this paper, we present two different methods for computing averages of strings, as well as an on-line version of the self-organizing map for strings. Both methods for computing averages are faster than the original one used by Kohonen and Somervuo, and one of them is suitable for on-line computation.

Keywords—**Self-organizing map; String average; String clustering.**

I. Introduction

Self-organizing maps are usually used for mapping complex, multidimensional numerical data onto a geometrical structure of lower dimensionality, like a rectangular or hexagonal two-dimensional lattice. The mappings are useful for visualization of data, since they reflect the similarities and vector distribution of the data in the input space. Each node in the map has a reference vector assigned to it. Its value is a weighted average of all the input vectors that are similar to it and to the reference vectors of the nodes from its topological neighborhood.

For numerical data, average and similarity are easily computed: for the average, one usually takes the arithmetical mean, and the similarity between two vectors can be defined as their inverse distance, which is most often the Euclidian one. However, for non-numerical data – like symbol strings – both measures tend to be much more complicated to compute. Still, like their numerical counterparts, they rely on a distance measure. For symbol strings one can use the *Levenshtein distance* or *feature distance*.

Having chosen the distance measure, one can define the average of a set of strings as, simply speaking, the string with the smallest distance from all strings in the set. The similarity can be again defined as inverse or, even better, negative distance between the strings.With those two measures and substituting reference vectors by reference strings, one can construct self-organizing maps of symbol strings.

Finding the average string of a string set can be computationally intensive: one usually has to search through a large number of strings and compare them with each one from the set, in order to find the best choice. We have developed an algorithm that checks only the best candidates, thus gaining much on speed. We have also developed a fast algorithm that incrementaly approximates the average, which finds its application in an on-line version of string SOM.

II. A brief description of the self-organizing maps for symbol strings

The self-organizing map of symbol strings [2] (*string SOM* for short) doesn't differ much from ordinary "numerical" SOM. It is also a low dimensional lattice of neurons (usually two-dimensional quadratic or hexagonal lattice, sometimes one- or three-dimensional), but instead of having a reference vector of input space dimensionality assigned to each node, reference strings are used. In the ordinary SOM, the reference vectors approximate the average of similar input vectors and input vectors similar to the reference vectors of the nodes from the topological neighborhood. In string SOM, the reference strings aproximate the averages of corresponding input strings.

Average can generally be defined for all types of data for which a distance function exists. Distance on a set E is defined as a function $d : E \times E \to \mathcal{R}$ such that

1. $d(x, x) = 0, \forall x \in E$
2. $d(x, y) > 0, \forall x, y \in E | x \neq y$
3. $d(x, y) = d(y, x), \forall x, y \in E$
4. $d(x, y) \leq d(x, z) + d(y, z), \forall x, y, z \in E$

For real vectors, the prefered distance measure is Euclidian distance. It is, however, not applicable for strings. A symbol string cannot be represented by a numerical vector. Although one can assign a numerical value – a *code*

– to a symbol, a coding in which numerical diferences between the codes reflect dissimilarities among corresponding symbols is hard to achieve. And even if one would succeed in finding such a coding, or if one decides to neglect the similarities between single symbols, a string of symbols cannot be acceptably represented by a vector of their corresponding numerical codes. The problems arise when one tries to compare strings of different lengths, as well as when one string is derived from another by insertion or deletion of symbols.

Fortunately, there are distance measures which work for symbol strings:

• *Levenshtein distance* is the minimum number of elementary transformations – insertion, deletion and substitution of a symbol – needed to transform one string into another [7]:

$$LD(s_1, s_2) = min\,(n_{ins} + n_{del} + n_{subst}) \qquad (1)$$

Closely related to it is *weighted Levenshtein distance* (WLD) [4], also known as *edit distance* [13], where different costs are assigned to each edit operation. The term *edit distance* is sometimes used only for the special case, where insertion and deletion have unit cost, and substitution is considered to be a deletion-insertion pair, having thus a cost of 2 [12].

• *Feature distance* [5] is the number of features in which the two strings differ. For strings, N-grams (substrings of N consecutive symbols) are the usual choice for features. If one string is longer than the other, the unmatched N-grams are also counted as differences:

$$FD(s_1, s_2) = max(N_1, N_2) - m(s_1, s_2) \qquad (2)$$

where N_1 and N_2 denote the number of N-grams in strings s_1 and s_2 and $m(s_1, s_2)$ is the number of matching N-grams.

The *Levenshtein distance* leads in general to a slightly better classification accuracy than the *feature distance*, but the latter has one big computational advantage when applied to self-organizing maps: searching and comparing based on it can be performed much faster than when using *Levenshtein distance* [6], [2].

Based on the distance measure, one can define an average as generalized median (the item having minimum sum of distances over the whole set) or generalized mean (the item having minimum sum of squared distances over the whole set). For similarity, negative distance is a better choice than the inverse, because it is defined even for items with zero distance. There are also other possibilities; for the learning algorithm it is only important that greater distance implies lower similarity and vice versa.

Both mean and median of a string set can be computationally intensive to find. The exhaustive search through all possible strings generally takes prohibitively long time. Still, some kind of search has to be performed in order to find the best choice. The algorithm by Kohonen and Somervuo starts by finding the *set average* (the set element having the smallest sum of distances or sum of squared distances to all other members of the set) as the initial guess for the average. This string is then systematically modified by inserting, deleting and replacing symbols at all positions in it, and the modification is accepted if it leads to a better average.

One notes that this is an off-line algorithm: the complete set has to be accessible in order to compute the average. Consequently, the string SOM has to be defined as off-line (batch) process. The algorithm for computing the map is roughly as follows: First, the map is initialized using Sammon projection [10]. Then, for each node in the map, a set is formed of all input strigs to whom the reference string of the node is the nearest reference string. Finally, for each node the union of the sets belonging to the node and its topological neighborhood is produced, the average over it is computed and taken as the new value of the reference string of that node. The process repeats from the second step until one obtains a stable map.

III. A FASTER APPROXIMATION OF THE STRING AVERAGE

For practical purposes, the algorithm for computing string averages, as explained in the previous section, doesn't guarantee finding the average. As long as one varies only one symbol at a time, it can get stuck in a local optimum. To leave it (i.e. to further decrease the sum of distances or their squares), one would have to change two or more symbols simultaneously. Increasing the number of simultanous variations in the algorithm would only slightly increase the likelihood of finding the global optimum: if the algorithm performs n simultanous changes, it can still happen that $n + 1$ are needed to leave the local optimum. But, the computing time would increase exponentially.

To us there is no known method of finding the average of strings in polynomial time. But we have developed an algorithm that also finds good approximations of average and that is faster than the original one presented above, especially for long strings. It is based on the following consideration:

Let us define the average of a string set as the median over that set, with Levenshtein distance as distance measure. Then we can interprete the average as the string from which all strings in the set can be produced with, on average, smallest number of edit operations. Now let us take a random string as an initial guess for the average and compare it with all strings in the set. Each

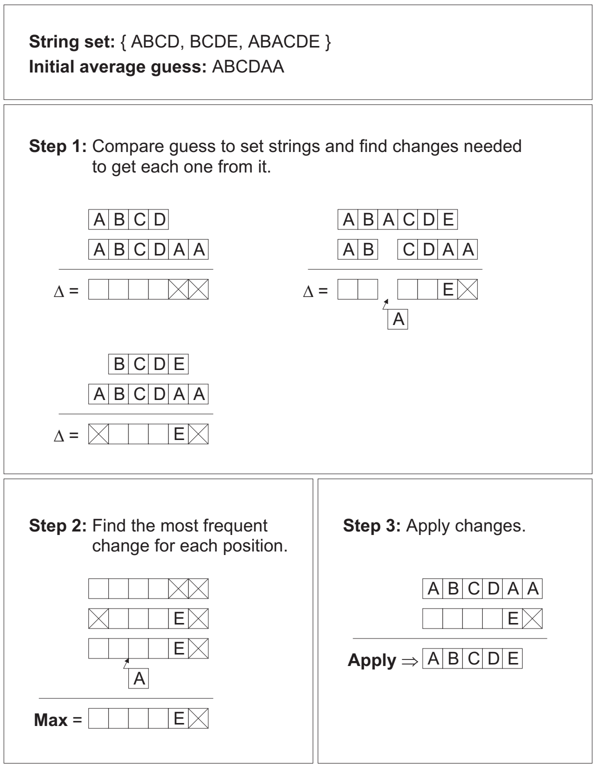

Fig. 1. An example for computing average of a string set.

comparison produces a set of edit operations (transformations) needed to obtain that string from the presumed average and that transformations can take place on different positions in the "average". For easier explanation, we shall, in addition to the three elementary transformations (insert, delete and substitute) introduce the "do nothing" as the fourth possibility. Now we consider all the transformations (resulting from all comparisons) that refer to a same position in the "average". On that single position we have a number of insertions (possibly of different symbols), replacements (also with different symbols), deletions and "do nothing" operations. It is easy to see that the transformation that appears most often is the optimal one, at least locally. If we would apply it, we would get in the next round of comparisons "do nothing" as the most frequent transformation for that position; in other words, the overall number of variations, excluding "do nothing", would decrease at that position, therefore decreasing the sum of distances. A simple example is shown in figure 1.

Since we perform the transformation of a symbol at each position independently of all other positions, there is no guarantee that applying all transformations leads directly into the global optimum. Insertion or deletion of a symbol at a position can have side effects on other positions, so several iterations may be needed. Also, the resulting average depends on the initial guess. Nevertheless, taking the set median (the set member with smallest sum of distances from other strings) as initial guess and taking the result of each iteration as the guess for the next one leads generally to a good aproximation of the average string after only few iterations.

More formally we can summarize the algorithm as follows:

1. Find the set median of the string set and use it as initial average guess.
2. Go through all strings in the set and find the transformations needed to get that string from the average guess.
3. For each position in the average guess, find the most frequent transformation (counting "do nothing" also as a transformation) and apply it to the average guess.
4. Take the result as a new guess average and repeat from 2 until no further improvement takes place.

Transformations needed to get one string from another are computed using dynamic programming [13]. For strings of approximately same length, the computing time increases quadratically with string length [11], although under certain assumptions faster algorithms are known [8], [9], [1]. The time needed to determine the best transformation for each position depends on the number of possible transformations, which is itself limited by the alphabet size. Since this has to be done for the whole set, this algorithm is approximately of $O(sl^2 + sla)$ complexity, l, s and a being string length, set size and alphabet size, respectively.

On the other hand, finding distances between two strings is generally of quadratic complexity, too. In the original algorithm, one has to compute them for every variation at every position in the string and then compare the resulting string with all others from the set. Consequently, the original algorithm is generally of $O(asl^3)$. When *feature distance* is used, one can apply redundant hash addressing in computing distances, thus reducing the complexity to roughly $O(al^3 + alsl')$, where l' denotes the average number of unique N-grams in strings. For short strings it can be approximated with l, since the probability that the same N-gram appears more than once in a string is negligible. But, for long strings over short alphabets, it is better approximated by a^N. For such strings, one can even put all unique N-grams of the test string into hash, reducing complexity to approximately $O(al^2l' + alsl')$. However, our experiments show that the constant overhead introduced by this hashing tends to overweight its benefits if the strings are not very long.

Execution time measurements on artificially generated data sets are shown in figures 2 to 7. For self-organizing maps, we used a 10×10 map, initialized with Sammon projection of a random set sample, and data sets were

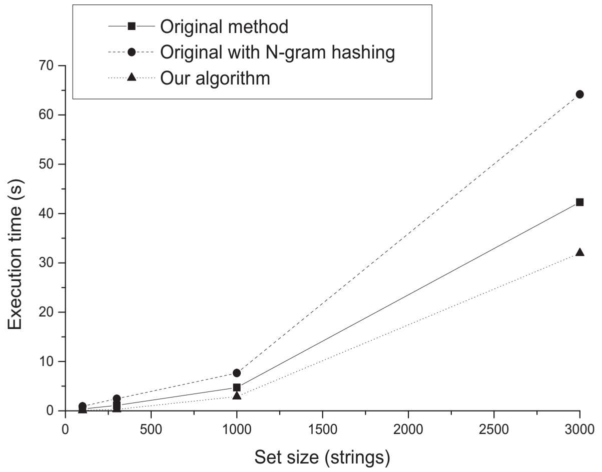

Fig. 2. Execution time for computing string average as a function of set size. Average string length = 12, alphabet size = 26.

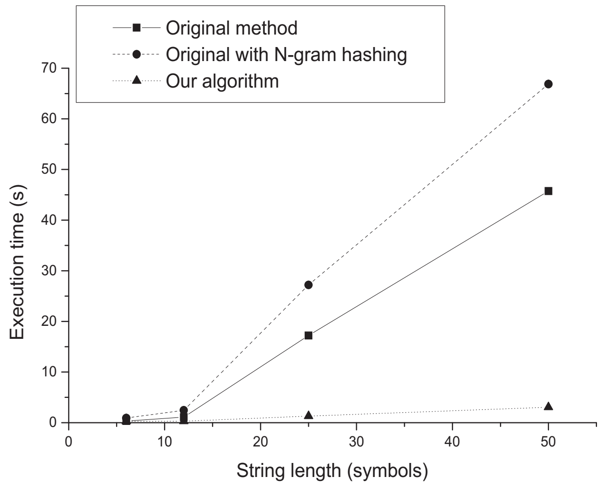

Fig. 3. Execution time for computing string average as a function of string length. Set size = 300, alphabet size = 26.

generated by introducing artificial errors to ten random words. Initial neighborhood radius was 1.5 and radius reduction factor was 0.95. All runs consisted of 10 iterations.

The algorithm presented above is also an off-line one. But the basic idea from it will help us to develop an on-line version.

IV. Computing string averages on the fly

In the on-line version of numerical SOMs [3], the average is approximated by correcting a guessed average by a portion of the distance between it and a input vector in every iteration:

$$\bar{x}(n+1) = \bar{x}(n) + \eta \left[x(n+1) - \bar{x}(n)\right] \qquad (3)$$

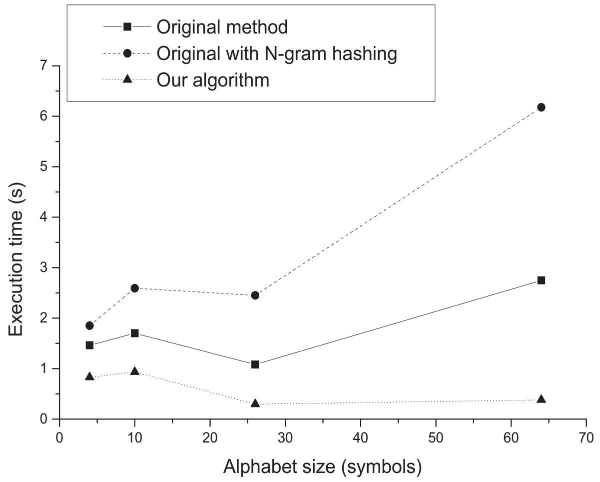

Fig. 4. Execution time for computing string average as a function of alphabet size. Average string length = 12, set size = 300.

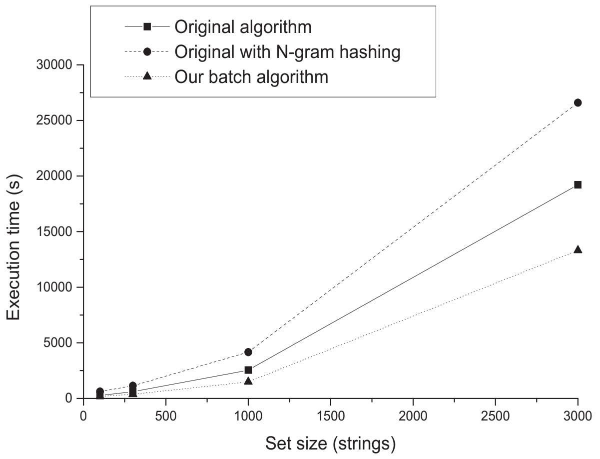

Fig. 5. Execution time for computing string SOM as a function of set size. Average string length = 12, alphabet size = 26.

where $\bar{x}(n)$ and $\bar{x}(n+1)$ denote approximations of the arithmetic mean in the n-th and $n+1$-th iteration, respectively, $x(n+1)$ the input vector in the $n+1$-th iteration, and η the correction factor.

One notes that, taking $\eta = \frac{1}{n+1}$ would lead to the correct arithmetic mean. In SOM, η is usually taken as a small number that further decreases from iteration to iteration.

Discrete values, like symbols, cannot be incrementally changed by only a small portion of error. However, one can accumulate errors and make a discrete change as soon as it reaches a certain level. This fact, combined with the algorithm presented in the previous section, leads to the algorithm for computing string averages on-line.

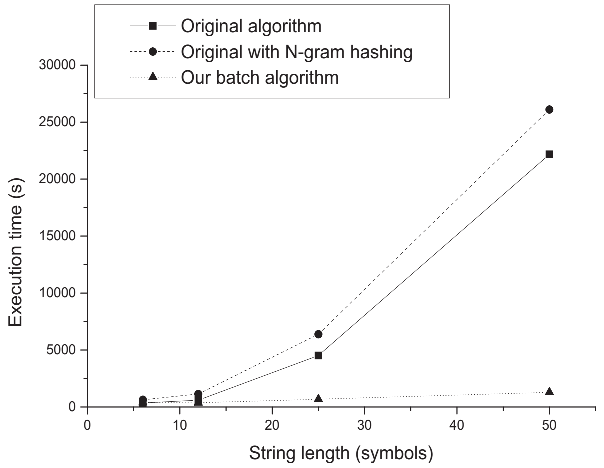

Fig. 6. Execution time for computing string SOM as a function of string length. Set size = 300, alphabet size = 26.

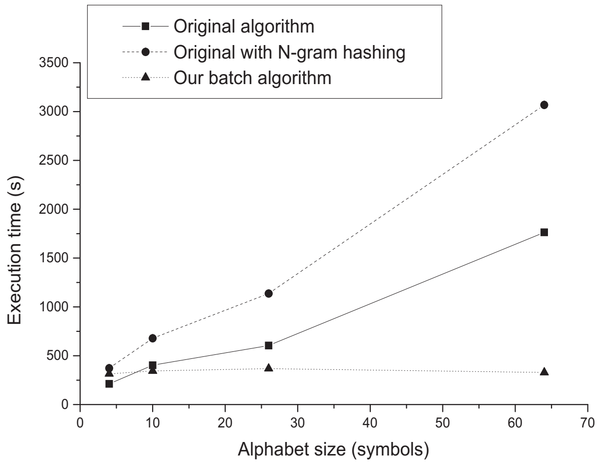

Fig. 7. Execution time for computing string SOM as a function of alphabet size. Average string length = 12, set size = 300.

A. Simple on-line averages

As in the numeric version, the first input string is taken as the first approximation of the average. Two counters are assigned to each symbol position in the average: one for number of comparisons since the symbol was last changed, and one for number of comparisons which resulted in a request for change. Then, as each new string appears on the input, the transformations needed to get it from the average are computed. The counters are correspondingly updated: the comparison counter is unconditionally incremented and the request-for-change counter is incremented if a transformation at that position is needed. If the request-for-change counter reaches a certain limit value, the transformation is applied to that position and both counters are reset.

The limit value, which triggers a change at a certain position, depends on the number of comparisons up-to-date and is computed statistically. Ideally, we would like to apply the change that is requested by most comparisons, as in the off-line version. Unfortunately, since the number of all possible changes for each position can be large, that would impose a need for many counters in order to keep track of their frequencies..

We shall use a simpler, although not quite correct variant of the decision procedure from the off-line version: a change to the symbol at a certain position should be made if more than the half of all strings differ at that position. Since the strings appear stochastically at the input, the statement "*more than the half of all strings differ at a position*" can be made only with certain statistical significance. The change is then applied to a symbol if the probability that the statement is not true is lower than the significance level.

Let α denote our significance level, n the number of comparisons at a string position and r_α the number of requests for change for that position. We shall suppose that the probability that a comparison results in a request for change is constant. Then, r_α is computed as the smallest integer for which the following inequality is satisfied:

$$\sum_{i=r_\alpha}^{n} \binom{n}{i} \left(\frac{1}{2}\right)^n \leq \alpha \qquad (4)$$

In practice, r_α can be well implemented as a look-up table, thus speeding up the computation. So, if a the request-for-change counter reaches r_α for chosen α and given value of comparison counter, a change of the symbol at that position is made.

Up till now we haven't said *which* change to make at the position. As mentioned above, we don't trace which request for change appeared most frequently. However, the request that appears most frequently has the highest probability to be the one that caused the counter to reach the limit. So the most simple method is to apply the current change to the symbol. If this decision proves wrong, further comparisons will cause again requests for change. Often it takes only few iterations to reach the stable state at a position.

Altogether, the algorithm consists of the following steps:

1. Take the first input string as the average guess and reset comparison and request-for-change counters.
2. Take the next symbol from the input and compute the transformations needed to obtain that string from the average guess.
3. For each position in the average guess, increment comparison counter and, if the comparison resulted in a transformation at that position, increase the request-for-change counter, too.

4. If the request-for-change counter has reached the limit value r_α, apply the transformation at that position and reset both counters.
5. Repeat from 2 as long as there are strings at the input.

The algorithm presented is also an approximative one. Although it leads to slightly inferior results compared to the off-line algorithms presented above, it is much faster, sometimes even several orders of magnitude, and therefore still very attractive for practical use. However, for larger sets consisting of strongly dissimilar strings, the algorithm can oscilate, never reaching the stable state. A simple modification brings significant improvement.

B. On-line averages – more sophisticated

The changes that the above algorithm makes to symbols are not the optimal ones. To find the optimal transformation, we shall, as in the off-line version, trace not only *if* a change is needed at a position in string, but also *which* changes are requested. When the most frequent request reaches the limit value, the change is performed.

The values of the request-for-change counters are distributed multinomially. But, conditonal distribution of a single counter is binomial. So, for the limit value, one can still use the simple binomial formula (4). Putting n to be the sum of the two most frequent requests for changes, we test if the most frequent request appears significatnly more often than the next most frequent.

As already mentioned above, this algorithm needs many counters, one for each type of request for change. This introduces also a certain computational overhead. Nevertheless, the benefits of this improvement – more in result quality than in computation time – justify additional effort.

V. ON-LINE STRING SOM

Based on the on-line algorithm for computing averages of strings, we have developed the on-line version of the self-organizing map of symbol strings. The adaptation algorithm for the string SOM resembles the one for numerical map:

1. Initialize the map with random strings from input and reset counters for reference strings
2. Take the next symbol from the input and find the node in the map with the nearest (most similar) reference string assigned to it
3. Compute the difference, i.e. the transformations needed to get the input string from the reference string
4. For each position in the reference string, increment comparison counter and, if comparison resulted in a

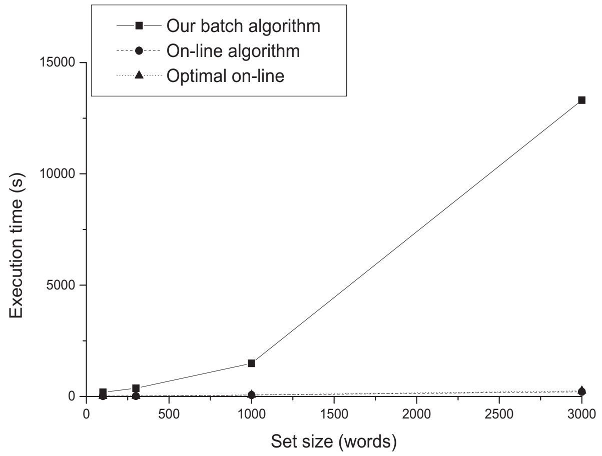

Fig. 8. Execution time for computing string SOM as a function of alphabet size. Average string length = 12, set size = 300.

transformation at that position, increase the request-for-change counter, too
5. If the request-for-change counter has reached the limit value r_α, apply the transformation at that position and reset both counters
6. apply steps 3, 4 and 5 to reference strings assigned to the topological neighborhood of the node
7. Repeat from 2 as long as there are strings at input or until a stable map is reached

The node with the nearest reference string can be found by searching for it through the whole map. It is, however, much more efficient to use *redundant hash addressing* [2]. This is not to be confused with the use of hash in computing the average, as mentioned in section III: in finding only the nearest string, one tries to avoid checking many strings, whereas in computing averages one is bound to check the whole set.

The hash is based on feature distance, whereas the algorithm for computing averages tends to minimize the Levenshtein distance. This inconsistency is, however, more of theoretical nature. Comparisons using Levenshtein and feature distance lead to very similar, if not same results and the reference strings generally differ enough to prevent assignment of input string to a false reference string. Thus, using hash can accelerate the computation for large maps without noticable influence on accuracy.

A comparison in execution time for batch and on-line string SOMs is given in figures 8 to 10. Whereas for the batch algorithm the number of adaptation cycles was always 10, for the on-line algorithms it depended on set size. That reflects the fact that in in the batch version, the whole map is updated in a single cycle, whereas the on-line algorithms update only a single node per cycle. For the set sizes of 100, 300, 1000 and 3000, the number of

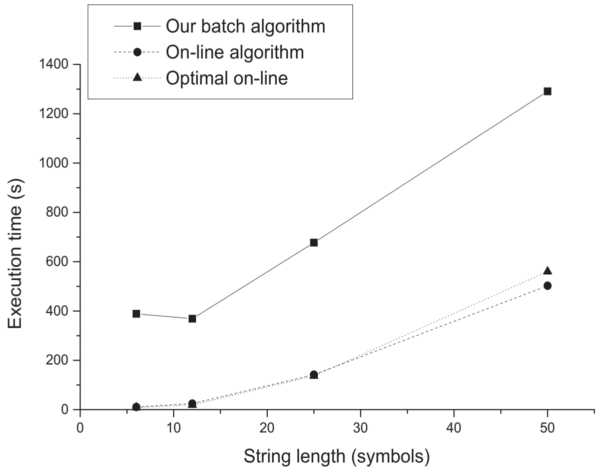

Fig. 9. Execution time for computing string SOM as a function of alphabet size. Average string length = 12, set size = 300.

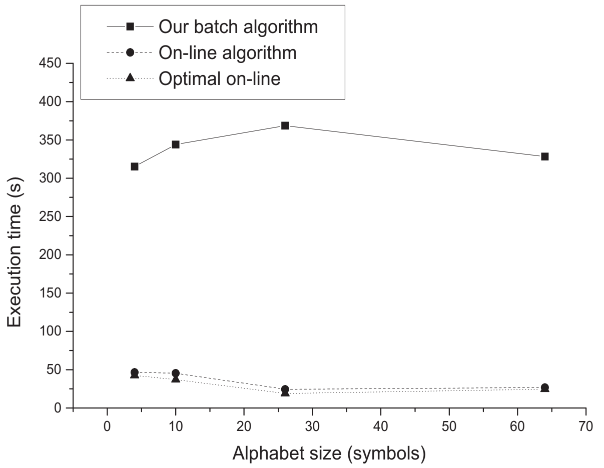

Fig. 10. Execution time for computing string SOM as a function of alphabet size. Average string length = 12, set size = 300.

adaptation cycles was set to 10000, 30000, 100000 and 300000, respectively.

VI. Examples of the string SOM

For an example application of the string SOM, we generated a set of 500 strings by intorducing noise to 8 english words: *always, certainly, deepest, excited, meaning, remains, safety,* and *touch*, and initialized a 10×10 quadratic map with the Sammon projection of a random sample from the set.

Figure 11 shows the initial map and the map after 10 cycles of training with our batch algorithm. We used an initial neighborhood radius of 1.5 and a radius reduction factor of 0.9. The complete training process took about 90 s on a Pentium II 300 MHz machine.

In figure 12 we show maps of same data, trained by our on-line algorithms. The initial neighborhood radius was set to 2.5 and the radius reduction factor 0.99995. 10000 training cycles took about 11 and 13 seconds for the simple and the improved algorithm, respectively.

VII. Conclusion

We have presented a faster batch algorithm and two variants of an on-line algorithm for computing averages of symbol strings. Together with the concept of similarity between strings, those algorithms build the basic blocks for self-organizing maps of symbol strings. Like their numeric counterparts, string SOMs are useful for visualizing similarities between patterns. The on-line algorithms are noticably faster than the off-line, and can generally be considered a better choice.

Besides for visualization, we believe that the concepts of average and simiarity for symbol strings can be applied in a wide range of problems concerning machine learning, clustering and pattern recognition of symbolic data. Possible applications include speech recognition, molecular biology and social sciences. Therefore we expect furher developments in this area.

VIII. Acknowledgments

We would like to thank our colleagues Boris Hollas and Achim Lilienthal for their helpful suggestions regarding statistics used in our algorithms.

References

[1] Z. Galil, K. Park, *An improved algorithm for approximate string matching*, SIAM Journal on Computing, 19 (6) (1990) 989-999
[2] T. Kohonen, P. Somervuo, *Self-organizing maps of symbol strings*, Neurocomputing 21 (1998) 19-30
[3] T. Kohonen, *Self-Organizing Maps*, Springer Series in Information Sciences, vol. 30, Springer Berlin Heidelberg 1995.
[4] T. Kohonen, *Self-Organization and Associative Memory*, Springer Series in Information Sciences, vol. 8, Springer Berlin Heidelberg 1988.
[5] T. Kohonen, *Content-Addressable Memories*, Springer Series in Information Sciences, vol. 1, Springer Berlin Heidelberg 1987.
[6] T. Kohonen, *Median Strings*, Pattern Recognition Letters 3 (1985) 309-313
[7] L. I. Levenshtein, *Binary codes capable of correcting deletions, insertions, and reversals*, Soviet Physics–Doklady, 10 (8) (1966) 707-710
[8] W. J. Masek, M. S. Paterson, *A faster algorithm computing string edit distance*, Journal of Computer and System Sciences 20 (1980) 18-31
[9] E. Ukkonen, *Algorithms for approximate string matching*, Information and Control, 64 (1985) 100-118
[10] J. W. Sammon, Jr., *A nonlinear mapping for data structure analysis*, IEEE Transactions on Computers, 18 (5) (1969) 401-409
[11] J. C. Setubal, J. Meidanis, *Intorduction to Computational Molecular Biology*, PWS Publishing Company 1997.
[12] G. A. Stepen, *String Searching Algorithms*, World Scientific, 1994.
[13] R. A. Wagner, M. J. Fischer, *The string to string correction problem*, Journal of the ACM 21 (1974) 168-173

zcertainly	reqmanj	certapinlj	rertaicly	eczten	excifzd	delhzsd	neepeksat	snumais	gstafetve
dlrtainfy	qoucuh	safety	safetz	afetuy	remizq	remirr	zafatv	rpeainx	oeteinly
exciutmehd	excqitd	safeuy	xied	pcied	excited	rmxns	remhins	dneebsm	ftetadtlcj
exuitld	tjwh	exied	xcimed	touco	emans	remins	cccited	remaxns	hexcitep
kfty	uuch	etailk	ssfty	afty	emainm	alapj	eeepet	fsocj	lepesw
rdemaina	fouzh	taaey	vxcited	aluas	alays	eepest	exuhibd	erainlvg	exdcistd
remainit	oulil	toulh	odch	acxas	alwaas	always	altays	aqways	ccepeqt
meafaninq	certkinrx	deanig	denaet	alwayr	ajlwxys	alwayv	eixciteo	alwasmys	cedtcitnlh
kertinuu	isafetag	lmunig	menibg	meanig	depqxt	depeaft	pdeeest	ertaqinsly	decejst
dsrjfemty	cemrtainy	meanitng	mseanqng	criainldy	meaniing	meanigng	deejest	cemrainly	deepeseot

safety	safety	safety	safety	excited	excited	excited	remains	remains	remains
safety	safety	safety	safety	excited	excited	excited	remains	remains	remains
safety	safety	safety	safety	ecited	excited	remains	remains	remains	remains
touch	touch	touch	safey	safey	emains	remains	remains	remains	remains
touch	touch	touch	touch	alwas	always	always	remays	remains	remains
touch	touch	touch	touch	alwas	always	always	always	always	always
touch	touch	touch	touch	alwas	always	always	always	always	always
touch	touch	touch	touch	meanns	always	always	always	always	always
certaily	certiny	meanig	meaning	meaning	meaning	depest	depest	depest	depest
certainly	certainly	meaning	meaning	meaning	meaning	deepest	deepest	deepest	deepest

Fig. 11. Example of string SOM for a set of 500 strings. Above: initial map. Below: the map after 10 cycles of batch training.

safety	safety	safety	sawey	always	alwas	eircited	excited	excited	excited
safety	safety	safey	slwey	always	always	excitts	excited	excited	excited
safety	safety	seaey	alway	always	always	lways	emites	excind	excied
oufch	otcwh	kjey	aays	always	rlways	remais	remains	remains	remains
touh	touch	tquh	mkaizh	mlays	memains	remains	remains	remains	remains
touch	touch	touch	meanicg	meaing	memaing	remains	remains	remains	remains
tounch	touch	touch	maneing	meaning	meaning	mening	eeepist	eepest	deeess
certainly	certainly	certainly	ertanin	meaning	meaning	meening	eepest	deepest	deepest
certainly	certainly	certainly	cetainly	meaning	meaning	eeeng	deepest	deepest	deepest
certainly	certainly	certainly	ceainly	meaning	meaning	meeneng	depest	depest	deepest

certainly	ceainly	saafely	safety	eaited	excited	excited	eemaind	remains	remains
certainly	sainly	safety	safety	safetd	excitd	excind	remats	remains	remains
stfnly	safety	safety	safed	excied	excited	remtnd	remains	remains	remains
toecy	tety	safeh	safed	saied	emaes	remins	remains	remains	remains
touch	touch	touch	ouch	aiay	elaiys	alays	remains	remains	remains
touch	touch	touch	ouch	aluas	alays	always	always	aeways	remais
touch	touch	touch	ouyh	alwas	always	always	always	aeways	depest
tninh	tancg	meanih	mewing	always	always	always	dewayt	deepest	deeest
meaning	meang	meanig	mening	meanis	deways	depeayt	deeest	depest	deeest
meaning	meaning	meaning	meaning	mening	mepint	depest	deepest	depest	deepest

Fig. 12. String SOM trained with on-line algorithms. Above: using simple on-line average algorithm. Below: using improved algorithm.