Mining Numbers in Text Using Suffix Arrays and Clustering Based on Dirichlet Process Mixture Models

Minoru Yoshida[1], Issei Sato[1], Hiroshi Nakagawa[1], and Akira Terada[2]

[1] University of Tokyo, 7-3-1, Hongo, Bunkyo-ku, Tokyo 113-0033
{mino,sato,nakagawa}@r.dl.itc.u-tokyo.ac.jp
[2] Japan Airlines, 3-2, Haneda Airport 3-Chome, Ota-ku, Tokyo 144-0041 Japan
akira.terada@jal.com

Abstract. We propose a system that enables us to search with ranges of numbers. Both queries and resulting strings can be both strings and numbers (e.g., "200–800 dollars"). The system is based on suffix-arrays augmented with treatment of number information to provide search for numbers by words, and vice versa. Further, the system performs clustering based on a Dirichlet Process Mixture of Gaussians to treat extracted collection of numbers appropriately.

Keywords: Number Mining, Suffix Arrays, Dirichlet Process Mixture, Clustering.

1 Introduction

Texts often contain a lot of numbers. However, they are stored simply as strings of digits in texts, and it is not obvious how to treat them as not strings but numeric values. For example, systems that treat numbers simply as strings of digits have to treat all numbers "1", "2", "213", and "215" as different, or all of them in the same way (e.g., by replacing them with "0"). In this paper, we propose treating numbers more flexibly, such as *similar* numbers like "1" and "2" should be treated as the same, "213" and "215" should also be treated the same, but "1" and "213" should be treated as different. *Range of numbers* is a representation of number collections that is appropriate for this purpose. In the above case, the collection of "1" and "2" can be expressed by the range "1..2" and the collection of "213" and "215" can be expressed by "213..215". Not only it can represent a lot of numbers compactly, but also it covers the numbers similar to the given collections not found in the given collection.

We propose a system that provides the following two basic indispensable functions for treating a range of numbers as normal strings:

- the function to derive appropriate number ranges from a collection of numbers,
- the function to search texts by using number range queries.

M.J. Zaki et al. (Eds.): PAKDD 2010, Part II, LNAI 6119, pp. 230–237, 2010.

The former is to *find* the range of numbers inherent in a collection of numbers and the latter is to *use* the extracted number ranges for further processing.

For the former problem of finding number ranges, the system dynamically clusters the numbers in the search results based on the Dirichlet process mixture (DPM) [1] clustering algorithm, which can automatically estimate the appropriate number of clusters. Our DPM model for number clustering is a mixture of Gaussians [2], which is a very popular example of DPM models. Inference for cluster assignment for DPM models has been extensively studied for many previous papers, including MCMC [3], variational Bayes [4], and A* or beam search [5]. However, our task is somewhat different from the ones discussed in these papers, because our task is to derive appropriate *number ranges*, which require constraints to be put on the derived cluster assignments that each cluster must consist of contiguous regions, and it is unobvious how to incorporate them into existing inference algorithms. To the best of our knowledge, no previous studies have discussed how to derive such number ranges on DPM models.

For the latter problem of the number range search, we propose using *suffix arrays* for the basic index structures. We call the resulting index structure *number suffix arrays*. The following operations are possible on number suffix arrays.

TF calculation: obtaining the counts for the queries that contain the range of numbers.

Adjacent string calculation: obtaining the strings (or tries) next to the range of numbers.

Search engine providers are one of many groups that have extensively studied indexing for searching by range of numeric values. Fontoura et al. [6] proposed an indexing method with inverted-indexes to efficiently restrict a search to the range of values of some of the numeric fields related to the given documents. In particular, Google search ("search by numbers") in English provides a search option ".." to indicate the number ranges. The inverted-index based methods for number range retrieval are for returning the *positions* of the numbers in the given range. On the other hand, our number suffix arrays not only return the positions, but also return the suffix array for the strings adjacent to the query, which can be used as a trie of the strings adjacent to the query and can be used for many text mining applications. These applications include extracting frequent adjacent string patterns for further text mining operations like synonym extraction (as shown in the later sections). In other words, number suffix arrays can be regarded as the extended version of the normal indexes for number ranges that are more appropriate for text mining tasks.

2 Number Suffix Arrays

The main component of our number mining system is *number suffix arrays*, which are based on suffix arrays [7] and can enable searches by numbers. Suffix arrays are data structures that represent all the suffixes of a given string. They are sorted arrays of (positions of) all suffixes of a string. By use of the suffix

array constructed from the corpus S, all the positions of any substring s of S can be obtained quickly (in $O(|s| \log |S|)$ time, where $|x|$ is the length of x) for any s by using binary search on the suffix array. Suffix arrays require $2|S|$ bytes[1] of additional space to store indices and even more space for construction. We assume that both the corpus and the suffix array are stored in memory. We denote by $S[i..]$ the suffix of S starting from index i.

Our algorithm for searching for a range of numbers is defined as follows. Assume that the input query is a string $s_1.[lb_1..ub_1].s_2.[lb_2..ub_2]...s_n$ where "." means concatenation of adjacent strings, and lb_k and ub_k are integers. Strings surrounded by `[` and `]` in a query represent the range of numbers from the number on the left of `..` to the number on the right of `...` For the query q =`KDD-[2000..2005]`, s_1=`KDD-`, $lb_1 = 2000$, $ub_1 = 2005$, and s_2="" (null string), where $n = 2$. Setting the current index array $ca = sa$ (sa is a suffix array of the whole input document), our algorithm iterates the following steps for $k = 1, 2, ..., n$.

1. Search for string s_k on the array ca, and obtain the resulting range of indices $[x \ldots y]$. Create a new array sa_2 of strings adjacent to s_k by letting $sa_2[i] = sa[x + i] + |s_k|$[2]. Let $ca = sa_2$.
2. Search for all digits that follow s_k. This is done by searching for the index of the character `0` on ca, and obtain the resulting index i_1, and in the same way, searching for the index of the character `9` on ca, and obtain the resulting index i_2. For each $i_1 \leq j \leq i_2$, parse the consecutive digits in the prefix of $S[sa_2[j]..]$ (suffixes of S starting from the position $sa_2[j]$), and obtain the resulting value d. If $lb_k \leq d \leq ub_k$, add the index i_3 (i_3: index of the end of the digits) to a new array sa_3. [3]
3. Sort the array sa_3 according to the alphabetic order of $S[sa_3[j]..]$. Let $ca = sa_3$.

In general, the range $[i_1 \ldots i_2]$ in step-2 in the above algorithm will not be so large because it is only covers the suffixes that are at least preceded by s_1 and start with digits. However, if s_1 is null (*i.e.,* the query starts by the range of numbers such as `[100..200] years old`), the range $[i_1 \ldots i_2]$ will be considerably large (it will be the number of all numbers in the text), which means scanning the range will be prohibitively time-consuming. Our basic idea to solve this problem is to make an additional array, which we call a *number array*, that retains *numeric ordering*. The number array na for corpus S is the array of indices that all point to the start point of all consecutive digits in S. It is sorted by the numeric order of the numbers represented by the digits that start from each position pointed to by the indices, and we can find (ranges of) numbers by performing binary search on this array with numeric-order comparison.

[1] This is if each index is represented by four bytes and each character takes two bytes.

[2] Here, $|s|$ is the length of string s.

[3] We do not use an approach to modify the corpus by replacing numbers with one character representing the value because it reduces the system's ability in some cases, e.g., it will limit variety of possible values to $2^{sizeof(char)}$, disable digit-pattern-matching queries such as "Boeing 7*7", etc.

3 Number Clustering by Dirichlet Process Mixture Models

Search results for number suffix arrays also may contain numbers. Number suffix arrays can describe collections of different numbers by number ranges, i.e., by the smallest and largest values, such as "[20..50]" for the collection of 20, 23, 30, 35, 42, and 50. The problem here is that these values do not always appropriately represent the collection. For instance, expressing the collection $< 1, 2, 3, 4, 1000, 1001, 1002 >$ as [1..1002] loses the information that values in the collection are concentrated in two ranges (*i.e.*, [1..4] and [1000..1002]). This problem can be avoided by dividing the collection into clusters.

Clustering algorithms that need to set the number of clusters (*e.g.*, K-means) are not appropriate for our situation because the appropriate number of clusters is different for each collection of numbers. Of the clustering algorithms that do not need data on the number of clusters, we selected the DPM [1] clustering algorithm because it provides the principles to probabilistically compare clustering results even if the number of clusters differs among distinct clustering results.

Given the collection of numbers $x_1, \cdots, x_n$[4], assume there exists the hidden parameter θ_i for each number x_i. The Dirichlet process [8] is a distribution over distributions and generates a discrete (with probability one) distribution over a given set (which is all the real numbers in our case). Let G be a distribution drawn from the Dirichlet process. Each value θ_i is drawn from G, where each observation x_i is generated from a distribution with parameter θ_i: $G \sim DP(\alpha, G_0)$, $\theta_i \sim G$, and $x_i \sim f(\theta_i)$. The parameters of the Dirichlet process are base distribution G_0 and concentration parameter α.

The Dirichlet process can give the probability of *clusters of* θ_i when G is integrated out. Here, θ_i and θ_j (, and thus x_i and x_j) are in the same cluster if $\theta_i = \theta_j$. Let C_j be a cluster of indices $c_{j1}, c_{j2}, \cdots, c_{j|C_j|}$ so that $\theta_{c_{j1}} = \theta_{c_{j2}} = \cdots = \theta_{c_{j|C_j|}}$. We denote the collection of all C_j as C. Then, the probability of the collection of θ_i is given as $p(\boldsymbol{\theta}) = \frac{\alpha^{|C|}}{\alpha^{(n)}} \prod_{j=1}^{|C|} G_0(\theta'_j)(|C_j| - 1)!$ where $|C|$ is the number of clusters, $|C_j|$ is the number of observations in the jth cluster and $\alpha^{(n)} = \alpha(\alpha + 1)(\alpha + 2) \cdots (\alpha + n - 1)$. θ'_j is the value for the jth cluster (*i.e.*, $\theta'_j = \theta_{j1} = \theta_{j2} = \cdots = \theta_{j|C_j|}$).

We use a DPM of Gaussians (or, equivalently, the infinite Gaussian mixture [2]) model. In our model, both G_0 and f are assumed to be Gaussians, with (mean,deviation) being (μ_1, σ_1) for the former and (θ_i, σ_2) for the latter: $G_0 = \mathcal{N}(\mu_1, \sigma_1)$, and $f_i = \mathcal{N}(\theta_i, \sigma_2)$.[5]

[4] We use the logarithm of each number in search results as x_i, which is based on the observation that relative sizes are appropriate as similarities between numbers rather than as absolute difference values.

[5] σ_1, σ_2 and μ_1 are fixed to reduce computation time. We set μ_1 to 0 and σ_1 to 100.0 to resemble the uniform distribution for the prior probability of θ_i to minimize bias in the value of θ_i. Other parameters are currently set to $\alpha = 1.0$ and $\sigma_2 = 1.0$.

The joint distribution of $\boldsymbol{x} = (x_1, x_2, \cdots, x_n)$ and $\boldsymbol{\theta}$ is thus

$$p(\boldsymbol{x}, \boldsymbol{\theta}) = \frac{\alpha^{|C|}}{\alpha^{(n)}} \prod_{j=1}^{|C|} (|C_j| - 1)! \frac{1}{\sqrt{2\pi}\sigma_1} exp\left\{ \frac{-(\theta'_j - \mu_1)^2}{2\sigma_1^2} \right\} \prod_{i=1}^{|C_j|} \frac{1}{\sqrt{2\pi}\sigma_2} exp\left\{ \frac{-(x_{c_{ji}} - \theta'_j)^2}{2\sigma_2^2} \right\}$$

We integrate out θ' because we need only cluster assignments, not parameter values themselves. This results in the the objective function to maximize (which indicates the goodness of clustering), which is denoted by $f(C)$.

$$f(C) = \frac{\alpha^{|C|}}{\alpha^{(n)}} \frac{1}{\sqrt{2\pi}^n \sigma_2^n} \prod_{j=1}^{|C|} g(C_j) \tag{1}$$

where

$$g(C_j) = (|C_j| - 1)! \frac{1}{\sqrt{1 + |C_j|(\frac{\sigma_1}{\sigma_2})^2}} exp\left\{ -\frac{1}{2\sigma_2^2} \left\{ (\sum_{i=1}^{|C_j|} x_{c_{ji}}^2) - \frac{\sigma_1^2}{\sigma_2^2 + |C_j|\sigma_1^2} (\sum_{i=1}^{|C_j|} x_{c_{ji}})^2 \right\} \right\}$$

The algorithm searches for the best cluster assignment that maximizes the objective function (1). Note that the objective function (1) is defined as the product of g for each cluster, which means that the function is "context-free" in the sense that we can independently calculate score $g(C_j)$ and then multiply it to calculate f because the value of $g(C_i)$ is not affected by changes in other clusters C_i s.t. $i \neq j$. Note that our purpose in clustering is to appropriately divide a given number collection into *continuous regions*. Therefore, we do not need to consider the case where a cluster is not a region (*i.e.,* elements in the cluster are separated by elements in another cluster, such as a case where $C_1 = \{1, 5\}$ and $C_2 = \{2, 6\}$.)

In this situation, the best cluster assignment can be found by a naive dynamic programming approach. We call this approach the *baseline algorithm* or *CKY algorithm* because it is a bottom-up style algorithm performed in the same way as the Cocke-Younger-Kasami (CKY)-parsing algorithm for context free grammar, which is popular in the natural language processing community.

In our approach, we accelerate the search further by using a greedy search strategy. Starting from no partition (*i.e.,* all elements are in the same region (cluster)), the algorithm divides each region into two sub-regions to best increase the objective function (1) and then recursively divides the sub-regions. If it is not possible to divide a region into two sub-regions without decreasing the objective function value, division stops.

More precisely, number clustering is done by calling the following function $partition(A)$, where A is the collection of all numbers input to the algorithm. After that, we obtain C as the clustering result.

Partition(N)**:** Find the best partition $left'(N)$ and $right'(N)$ that maximizes $g(left(N))g(right(N))$. If $\alpha \cdot g(left'(N))g(right'(N)) \leq g(N)$, then add N to C (*i.e.*, partitioning of N stops and N is added to the resulting cluster set). Otherwise, call $partition(left'(N))$ and $partition(right'(N))$ recursively.

Here, α is multiplied with $g(left'(N))g(right'(N))$ because partitioning increases the number of clusters $|C|$ that appear as $\alpha^{|C|}$ in objective function (1).

4 Experiments: Synonym Extraction

Our flexible number handling is useful in many text-mining tasks, especially when we want to use the numbers as some kind of *contexts*. A typical example is measuring the semantic similarities of words. When measuring word similarities, we typically calculate the *distributions* of words related to word w (e.g., distribution of words around w, distribution of words that have dependency relations with w, etc.) as the contexts of w, and measure the similarities of the meanings of the words w_1 and w_2 by calculating the similarities of their contexts.

A direct application of measuring similarities of words is *synonym extraction*. Especially, we developed an algorithm to dynamically extract synonyms of given queries using suffix arrays [9]. To find words similar to a given query q, the algorithm extracts context strings (*i.e.*, strings that precede or follow q) by using suffix arrays[6], which in turn are used to find strings surrounded by these extracted contexts.

We enhanced the algorithm by adding the ability to appropriately treat numbers in context strings in number suffix arrays. For example, we can use the context strings "[10..20] persons" to cover all numbers between 10 and 20 preceding the word "persons", while in naive suffix arrays, only raw strings such as "11 persons" and "17 persons" can be used as contexts. Our number suffix arrays can thus improve coverage of contexts and extracted collections of synonyms.

We evaluate the performance of synonym extraction with number suffix arrays to investigate whether number suffix arrays enhance text mining. We used aviation-safety-information texts from Japan Airlines that had been de-identified for data security and anonymity. The reports were in Japanese, except for some technical terms in English. The size of the concatenated documents was 6.1 Mbytes. Our text-mining system was run on a machine with an Intel Core Solo U1300 (1.06 GHz) processor and 2 GByte memory. All algorithms were implemented in Java. The size of the number array for each (normal or reversed) suffix array was 60,766.

To evaluate the performance of the system, we used a thesaurus for this corpus that was manually developed and independent of this research. The thesaurus consists of $(t, S(t))$ pairs, where t is a term and $S(t)$ is a set of synonyms of t. We provided t as a query to the system, which in turn returned a list of synonym candidates $\langle c_1, c_2, ..., c_n \rangle$ ranked on the basis of their similarities to the query. $S(t)$ was used as a correct answer to evaluate the synonym list produced by the system. The number of queries was 404 and the average number of synonyms was 1.92.

We compared the average precision [10] of our algorithm with the baseline (using naive suffix arrays) and the no-clustering version (using number suffix arrays

[6] We use two suffix arrays: one is a normal suffix array and the other is a *reversed suffix array*, which is a suffix array constructed from the reversed original text.

Table 1. Results of Synonym Extraction. σ_1 was set to 100.0. AvPrec is the average precision (per cent) and Time is the average execution time (per second) for each query.

Algorithm	AvPrec	Time
Baseline	40.66	2.294
No Clustering	41.30	10.272
Our Algorithm	41.68	7.837

Table 2. (Left:) Execution Time (Per Second) for Number Queries. NumStart is for Queries that Start with Number Ranges, and NotNumStart is for Queries that Start with Non-digit Characters. (Right:) Execution time (per second) and total log likelihood of number clustering.

Algorithm	NumStart	NotNumStart
Baseline	169.501	0.711
w/ Number Arrays	12.87	0.632

Algorithm	Time	Log Likelihood
CKY	102.638	-168021.7
Greedy	0.170	-168142.2

without number clustering). The results are shown in Table 1. We observed that the performance was improved by using number suffix arrays by about 0.6 percent, which was improved further by about an additional 0.4 percent by performing number clustering. However, the average execution time for each query became 3.5–4.5 times larger than that of the baseline. For practical use, we will have to reduce the execution time by reducing the number of range-starting queries (*i.e.*, the queries that start with a range of numbers).

4.1 Results: Speed and Accuracy of the Algorithm

We stored all the queries to the number suffix arrays and all the collections of numbers for number clustering that appeared in the above experiments. We randomly selected 200 queries that included the range of numbers for each suffix array (normal and reversed), resulting in 400 queries in total. Of each 200 queries, 100 started with a range of numbers (indicated as "NumStart"), and the remaining 100 started with non-digit characters (indicated as "NotNumStart"). We also randomly selected 1000 collections of numbers, and used them to measure the accuracy and execution time of our number clustering algorithms.[7]

The result of the query-time experiment is shown in Table 2 (left). We observed that the search time for queries starting with a range of numbers was drastically reduced by using the number arrays. Considering the large ratio of the search time of NumStart and NotNumStart, using the number arrays is an efficient way to conduct a number search.

The results of a comparison of two clustering algorithms are shown in Table 2 (right). The greedy algorithm was much faster than the baseline CKY algorithm. The important point here is that the difference of total log-likelihood values

[7] Only collections whose sizes were from 50 to 1000 were selected. The average size of collections was 98.9.

between the greedy (approximate) algorithm and the baseline was quite small, which suggests that using the greedy algorithm for number clustering achieves much faster processing with almost no sacrifice of quality of the clustering results.

5 Conclusion and Future Work

We described number suffix arrays, which enable us to search for numbers in text. The system is based on suffix arrays and DPM clustering. We also showed applications of number suffix arrays to text mining, including synonym extraction, where the performance could be improved by using number suffix arrays. Future work includes developing more sophisticated preprocessing for numbers such as normalization of number expressions.

Acknowledgement

This research was partially supported by Fujitsu Laboratories Ltd.

References

1. Antoniak, C.E.: Mixtures of Dirichlet processes with applications to Bayesian nonparametric problems. The Annals of Statistics 2(6), 1152–1174 (1974)
2. Rasmussen, C.E.: The infinite Gaussian mixture model. In: Advances in Neural Information Processing Systems, 13th Conference, NIPS 1999, pp. 554–560 (2000)
3. Jain, S., Neal, R.M.: Splitting and merging components of a nonconjugate dirichlet process mixture model. Technical Report 0507, Dept. of Statistics, University of Toronto (2005)
4. Blei, D.M., Jordan, M.I.: Variational inference for dirichlet process mixtures. Bayesian Analysis 1(1), 121–144 (2006)
5. Daumé, H.: Fast search for Dirichlet process mixture models. In: Proceedings of the 11th International Conference on Artificial Intelligence and Statistics (AISTATS), pp. 83–90 (2007)
6. Fontoura, M., Lempel, R., Qi, R., Zien, J.Y.: Inverted index support for numeric search. Internet Mathematics 3(2), 153–186 (2006)
7. Manber, U., Myers, G.: Suffix arrays: A new method for on-line string searches. In: Proceedings of the first ACM-SIAM Symposium on Discrete Algorithms, pp. 319–327 (1990)
8. Ferguson, T.S.: A Bayesian analysis of some nonparametric problems. The Annals of Statistics 1(2), 209–230 (1973)
9. Yoshida, M., Nakagawa, H., Terada, A.: Gram-free synonym extraction via suffix arrays. In: Li, H., Liu, T., Ma, W.-Y., Sakai, T., Wong, K.-F., Zhou, G. (eds.) AIRS 2008. LNCS, vol. 4993, pp. 282–291. Springer, Heidelberg (2008)
10. Chakrabarti, S.: Mining the Web: Discovering Knowledge from Hypertext Data. Morgan-Kaufmann Publishers, San Francisco (2002)