Succinct BWT-based Sequence prediction

Rafael Ktistakis[1], Philippe Fournier-Viger[2], Simon J. Puglisi[3], and Rajeev Raman[1]

[1] Department of Informatics, University of Leicester, UK
[2] Harbin Institute of Technology (Shenzhen)
[3] Department of Computer Science University of Helsinki, Finland
{crk15,r.raman}@leicester.ac.uk, philfv@hit.edu.cn, puglisi@cs.helsinki.fi

Abstract. Sequences of symbols can be used to represent data in many domains such as text documents, activity logs, customer transactions and website click-streams. Sequence prediction is a popular task, which consists of predicting the next symbol of a sequence, given a set of training sequences. Although numerous prediction models have been proposed, many have a low accuracy because they are lossy models (they discard information from training sequences to build the model), while lossless models are often more accurate but typically consume a large amount of memory. This paper addresses these issues by proposing a novel sequence prediction model named SuBSeq that is lossless and utilizes the succinct Wavelet Tree data structure and the Burrows-Wheeler Transform to compactly store and efficiently access training sequences for prediction. An experimental evaluation shows that SuBSeq has a very low memory consumption and excellent accuracy when compared to eight state-of-the-art predictors on seven real datasets.

1 Introduction

Sequences of symbols (strings) are a type of data found in many domains. For instance, they can be used to represent sequences of words in a text, events in a business process log, purchases made by customers, or point-of-interests visited by tourists. An important task in data mining is sequence prediction. Given a multi-set of training *strings* (or sequences) $\hat{\mathsf{D}} = \{\mathsf{x}_1, \ldots, \mathsf{x}_d\}$ defined over a finite ordered *alphabet* of symbols, sequence prediction consists of predicting the next symbol of the prefix of an unknown query sequence Q. The underlying assumption is that all the strings are created by a same underlying process. To perform sequence prediction, a predictor can be trained using the training strings. Then the predictor can perform predictions.

Various sequence prediction models have been proposed, having various characteristics. They have been used in many domains to perform tasks such as predicting heart failure [18], human activities [19] and webpage prefetching [6]. Although numerous prediction models have been proposed, many are lossy models [3,9,15,16,21]. In other words, they discard information from training sequences to build small models. But the drawback of this approach is that they

may lack information when its time to make a prediction, which can result in low prediction accuracy [7]. Some models such as DG [15] also adopt simplifying assumptions such that each symbol of a string only depends on the previous one. But this assumption often does not hold in real life applications.

The aforementioned limitations of lossy predictors have recently been addressed by proposing lossless models, which keep all information about training sequences in memory to perform more accurate predictions. The assumption is that a lossless model should be more accurate because they can use all the available information to make each prediction. Some of the best models of this type is CPT [7], which was then extended as CPT+ [6]. These models store training sequences in a trie-based structure, and were shown to be more accurate than multiple state-of-the-art lossy models. However, the CPT/CPT+ have several important drawbacks:

- To perform a prediction, the CPT/CPT+ models utilize the bag-of-words model, which does not consider the order between symbols. But for some domains, the order is important.
- The CPT/CPT+ models require choosing several dataset-specific parameters. The prediction accuracy can vary greatly depending on how these parameters are set. Setting these parameters is not trivial and requires to have background knowledge or use a trial-and-error approach to find optimal parameter settings.
- All lossless predictors end up storing the entire training sequence in main memory. Thus, it is essential that a lossless predictor should store the training sequence *space-efficiently*. We use the following variables to denote the size of the sequence database D: d is the number of sequences, M is the total length of all the sequences and σ is the alphabet size. We note that the information-theoretic lower bound for storing D is $M \log \sigma$ bits[4] in the worst case. On the other hand:
  - CPT+ uses σ bit-strings of length d to represent the sets of symbols contained in each sequence. This alone takes $d\sigma$ bits, which can be much larger than $M \log \sigma$ bits if σ is large.
  - CPT+ stores the training dataset in a trie. In the worst case, there could be $\Omega(M)$ trie nodes, and each trie node contains three (64-bit) pointers, a significant overhead.
  - CPT+ uses ideas such as Patricia compression and replacing frequently occurring sub-sequences by a single symbol to try to minimize the number of trie nodes[6]. However, success is unpredictable, and the frequent pattern mining slows down the training phase.
- During the prediction phase, given a query Q of k symbols, CPT+ performs several bitwise-and of up to k bit-strings of length d each to find sequences containing a subset of symbols in Q. This takes $O(f(k) \cdot d)$ time where $f(k)$ can be as large as 2^k. In practice, many fewer than 2^k combinations are tried, and the constants in the $O()$ are small. However, as we show, the query time of CPT+ grows linearly with d.

[4] Logs are to base 2 unless stated otherwise.

This paper addresses drawbacks of the CPT/CPT+ models by proposing a novel sequence predictor named SuBSeq. This model adopts the succinct Wavelet Tree data structure and the Burrows-Wheeler Transform to store training sequences in a very compact way, while still allowing fast access to training sequences for prediction. An experimental evaluation shows that SuBSeq has a very low and predictable memory consumption (the space usage varies between 1.6 and 2.2 times the binary size of D) and excellent accuracy when compared to state-of-the-art predictors on real datasets. Last but not least, SuBSeq is largely parameter-free.

The rest of this paper is organized as follows. Section 2 introduces preliminaries about sequence prediction. Section 3 presents the proposed SuBSeq predictor. Section 4 presents the performance evaluation. Finally, a conclusion is drawn and future work is discussed.

2 Preliminaries

Strings. A string $\mathsf{x} = \mathsf{x}[0..n-1] = \mathsf{x}[0]\mathsf{x}[1]\ldots\mathsf{x}[n-1]$ is a sequence of $|\mathsf{x}| = n$ symbols drawn from a constant ordered alphabet of size σ. For $i = 0, \ldots, n-1$ we write $\mathsf{X}[i..n-1]$ to denote the *suffix* of X of length $n-i+1$, that is $\mathsf{X}[i..n-1] = \mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[n-1]$. We will often refer to suffix $\mathsf{X}[i..n-1]$ simply as "suffix i". Similarly, we write $\mathsf{X}[0..i]$ to denote the *prefix* of X of length $i+1$. We write $\mathsf{X}[i..j]$ to represent the *substring* $\mathsf{X}[i]\mathsf{X}[i+1]\ldots\mathsf{X}[j]$ of X that starts at position i and ends at position j.

In this paper we consider a multiset of d strings $\hat{\mathsf{D}} = \{\mathsf{x}_1, \mathsf{x}_2, \ldots \mathsf{x}_d\}$. We represent $\hat{\mathsf{D}}$ as a single string by concatenating the strings in D into a single string $\mathsf{D} = \mathsf{x}_1\$\mathsf{x}_2\$\ldots\$\mathsf{x}_d$, using a special symbol \$ to delineate individual strings, which does not occur in any string x_i. We let $M = |\mathsf{D}|$ denote the length of D.

Suffix Arrays. We make use of several standard data structures built from D. The first of these is the suffix array [10], denoted SA, which is an array $\mathsf{SA}[0..M-1]$ containing a permutation of the integers $0..M-1$ such that $\mathsf{D}[\mathsf{SA}[0]..M-1] < \mathsf{D}[\mathsf{SA}[1]..M-1] < \cdots < \mathsf{D}[\mathsf{SA}[M-1]..M-1]$. In other words, $\mathsf{SA}[j] = i$ iff $\mathsf{D}[i..M-1]$ is the j^{th} suffix of D in ascending lexicographical order.

The Burrows-Wheeler Transform [2,11], denoted BWT is a string $\mathsf{BWT}[0..M-1]$ is a permutation of D defined by SA, such that $\mathsf{BWT}[i] = \mathsf{D}[\mathsf{SA}[i]-1]$, except when $\mathsf{SA}[i] = 0$, in which case $\mathsf{BWT}[i] = \mathsf{D}[M]$. See Fig. 1 for an example.

Backward Search. The FM-index is a compressed text index (see [13]) that consists of two main components: a wavelet tree build from the BWT string, and an array C of σ integers such that $\mathsf{C}[c]$ gives the total number of symbols in the BWT string that are less than symbol c. Searching with an FM-index is based on a procedure called *backward search*, which finds the range of SA containing all suffixes that begin with a given query pattern Q. This range then contains the positions of occurrence of Q in D. Figure 2 shows how backward search is used for counting the number of occurrences (the count query). In the algorithm, $\mathsf{C}[c]$ is the position of the first occurrence of the symbol c in F, and the function $\mathsf{rank}_{\mathsf{L}}$ is defined as $\mathsf{rank}_{\mathsf{L}}(c, j) \equiv \big|\{i \mid i < j \text{ and } \mathsf{L}[i] = c\}\big|$. The main difference between

the members of the FM-family is how they implement the $\mathsf{rank_L}$-function. The best ones use wavelet trees.

L	SA	
A	6	$
N	5	A $
N	3	A N A $
B	1	A N A N A $
$	0	B A N A N A $
A	4	N A $
A	2	N A N A $

Fig. 1: SA and BWT string L for string D = BANANA$.

Algorithm FM-Count(Q$[0..k-1]$)
1: $b \leftarrow 0$; $e \leftarrow M$
2: **for** $i \leftarrow m-1$ **downto** 0 **do**
3: $\quad c \leftarrow \mathsf{Q}[i]$
4: $\quad b \leftarrow \mathsf{C}[c] + \mathsf{rank_L}(c, b)$
5: $\quad e \leftarrow \mathsf{C}[c] + \mathsf{rank_L}(c, e)$
6: $\quad$ **if** $b = e$ **then break**
7: **return** $e - b$

Fig. 2: Counting pattern occurrences using backward search.

Wavelet Tree. The wavelet tree [12] of string D over an alphabet Σ is a binary tree with leaves labelled by the symbols of Σ. Each node v is associated with the subsequence of D consisting of those symbols that appear in the subtree rooted at v. The associated strings are not stored; instead each internal node v stores a bitvector $\mathsf{B}(v)$ that tells for each character in the associated string whether it is in the left or right subtree of v.

In a wavelet tree the total length of the bitvectors is $|\mathsf{D}|\lceil \log |\Sigma| \rceil$, which is exactly the length of D in bits using the standard representation.

A rank query $\mathsf{rank_D}(c, r)$ over a wavelet tree is evaluated by a traversal from the root to the leaf labelled by c. Wavelet trees answer rank queries in $O(\log \sigma)$ time. A similar procedure enables one to access a given symbol $\mathsf{D}[i]$ in $O(\log \sigma)$ time, or to enumerate all the distinct symbols in a range of the string, as well as compute the frequency of each of those symbols. Wavelet trees answer these $\mathsf{distinct}(i, j)$ queries in $O(k \log \sigma)$ time, where k is the number of distinct symbols in $\mathsf{D}[i..j]$. Wavelet trees also support the query $\mathsf{select}(c, i)$ in $O(\log \sigma)$ time, which returns the position of the ith occurrence of symbol c in D. The queries rank, select, access, and distinct involve rank (or select) queries over the bitvectors stored on the root-to-leaf path. There are many data structures for representing bitvectors so that rank and select queries can be answered in constant time [14,17]. These data structures are a standard component in succinct data structure design. Recent experimental studies of these bitvectors can be found in [5,8].

3 Succinct BWT-based Sequence prediction model

The *Succinct BWT-based Sequence prediction* model (SuBSeq) is a new lossless predictor. Its main distinctive characteristics are that (1) efficiently stores the entire input training data without any loss (2) fetches training sequences similar

to a given sequence (query prefix) (3) it does not depend in any parameter-set fine-tuning in order to be accurate (4) SuBSeq keeps into account the item order of a given query prefix. The latter is the main key difference to the CPT+ prediction model. CPT+ searches for sequences using the bag-of-words model. This model does not take into account the items order of a prefix for matching it in the training data (which might be important aspect for some domain applications, as discussed).

3.1 Algorithm description

The SuBSeq prediction algorithm is consisted of two main phases; the train phase and the ready-for-prediction phase. A multiset $\hat{\mathsf{D}}$ of training sequences is given as an input. During the train phase, SuBSeq will use the D to produce the FM-index and store BWT in memory using a wavelet tree. During the ready-for-prediction phase, SuBSeq is ready to answer query prefixes. The answers that SuBSeq returns can further be evaluated with the *query suffix* (see Section 4.2).

For every query prefix SuBSeq will try to give an answer by finding similar sequences in its training data sequences. This is done through the given query prefix and a generated collection of *sub-queries*. Due to the fact that SuBSeq is only able to locate exact matches of a given pattern in its training data, it is essential to have a mechanism that expands our prediction model coverage to more training data. The collection of sub-queries plays the role of this mechanism. Every sub-query comes from the initial query prefix. These are produced by allowing operations of deletion and substitution. The deletions are always at the start of the query or sub-query and the substitutions are limited to two.

Example. For a given $\mathsf{Q} = [a, b, c, d]$, SuBSeq will try to find exact matches for $\mathsf{Q}_1 = [a, b, c, d]$, $\mathsf{Q}_2 = [¿, b, c, d]$, $\mathsf{Q}_3 = [a, ¿, c, d]$, $\mathsf{Q}_4 = [a, b, ¿, d]$, $\mathsf{Q}_5 = [b, c, d]$, $\mathsf{Q}_6 = [¿, c, d]$, $\mathsf{Q}_7 = [b, ¿, d]$, $\mathsf{Q}_8 = [c, d]$, $\mathsf{Q}_9 = [¿, d]$

On the example above we denote with ¿ the place where we can replace with any symbol from our alphabet. Assuming our alphabet as $\Sigma = \{a, b, c, d\}$ then SuBSeq can match Q_6 with some example training sequences like: $[a, c, d, a, d]$, $[b, c, d, c, a]$, $[c, c, d, b, b]$, $[d, c, d, a, b]$.

After SuBSeq has found the similar sequences, it uses them to produce possible answers and eventually order them according to a *weight*. Producing possible answers is done through the *consequents* of the similar sequences. The consequent of a similar sequence s is considered the subsequence from the item common to both s and the current (sub-)query used, and up to the last item of s. For SuBSeq we will be using consequents of length up to two items long. Every time a (sub-)query is used to find similar training sequence, we come up with consequents. The items of the consequents are put into a *Frequency Array* and they are ordered by a weight. A final prediction answer is the item in the array with the highest weight value. The final answer is given either (a) when SuBSeq has collected all possible consequents for both the initial query prefix and its all produced sub-queries or (b) when a threshold of confidence is met.

Finally, when an item of a consequent is inserted to the frequency array, it is assigned a weight value. If the item exists in the array then the new value is added-up on the old value. The weight formula is defined as $w = y/Y + (2 - sub)/2 + 1 + r$. We consider y to be the suq-query length, Y the initial query length, sub the number of substitutions and $r = \frac{1}{index+1}$. The later indicates the index of the item in the consequent.

3.2 Implementation using FM-index

We mainly need four core functions; (1) ***backwardSearch*** (2) ***forwardSearch*** (3) ***neighbourExpansion*** (4) ***getConsequents***.

The **backwardSearch** can be implemented by tweaking the **FM-Count** (see Figure 2) to return the (b, e) for a query item at a time.

The **forwardSearch** does the opposite of the *backwardSearch* for a given i. It gives the index $i' = \mathsf{C}[c] + rank_{\mathsf{L}}(c, i)$ where $c = \mathsf{L}[i]$, and $c' = \mathsf{L}[i']$ occurs after c in D.

The **neighbourExpansion** constitutes the key function of our prediction model. Using the FM-index, one can only find exact matches for a given pattern. This creates a twofold issue; (1) there is no way to locate similar training sequences (2) usually in sequence prediction, searching only for exact matches does not give an enough coverage (if any) for confident predictions. The main idea of neighbour expansion is that for a given query prefix, it will perform a normal *backwardSearch* if the prefix does not have any substitutions in place or for any substitution that it mets it will recursively expand to all possible symbols that might follow. Taking into account our previous example of sub-queries, Q_3, we will make the following assumption; before a $[c, d]$ all of the $\{a, b, c, d\}$ appear in the training data. This can be figured out with a $\mathsf{distinct}$ call for a range in L. Then Q_3 will be expanded to $[a, a, c, d]$, $[a, b, c, d]$, $[a, c, c, d]$ and $[a, d, c, d]$ for a normal *backwardSearch* each.

The **getConsequents** utilises the $forwardSearch$ definition to obtain the consequents for ranges that have been acquired through the $neighbourExpansion$. Expanded sub-queries which result in patterns that have already been used, are excluded. We do this by utilising a bit-vector of length M. Every index of successful $neighbourExpansion$ ranges, is a set bit in the bit-vector. Thus, consequents from sub-queries that have been prior utilised, will not be re-used and only new consequent information will added in Frequency Array.

A C++ implementation of our prediction model can be found on `github.com/rafkt/SUBSEQ`.

4 Evaluation

We split this section as: the set-up environment, our experimental aims, the competition to our prediction model and finally the discussion of accuracy and performance evaluation. For this section, full details about our experimental data and about our results can be found on `github.com/rafkt/SUBSEQ`.

4.1 Experimental Setup

Environment. Experiments were performed under macOS 10.14.1 with an Intel Core i7 (4 Cores, 256KB L2 per Core, 8MB L3), 32GB DDR3 1867MHz RAM and a 8.0 GT/s Link speed SSD. The lossless predictors, CPT+, CPT, were ran using *IPredict* framework [6] under java version 1.8.0_112 with *JIT* enabled which allows the bytecode to be compiled into native machine code, allowing a fair comparison with native implementations. The SuBSeq Predictor was compiled under clang-1000.11.45.5, while SPiCe baseline [1] was compiled and run under Python 2.7.10. We used the *sdsl-lite* library [4] for implementing SuBSeq.

Aims. To measure and compare different prediction models in terms of their accuracy and their performance. Performance is measured in terms of the execution time a prediction model needs to train itself; the execution time it needs to complete answering a testing set; the memory usage it utilises after the training phase is complete.

Competition. We compare SuBSeq with a variety of state-of-the-art lossy and lossless predictors. These are: All-K-order Markov (AKOM) [16], LZ78 [21], Transition Directed Acyclic Graph (TDAG) [9], Prediction by Partial Matching (PPM) [3] and Dependancy Graphs (DG) [15]. We also included a spectral learning prediction model from SPiCe competition [1]. We also compare SuBSeq with CPT+ [6] as it is the current state-of-the-art lossless prediction model.

Data. For our experiments we used datasets with various characteristics from *SPMF* library[5] library. In addition, we used synthetic data[6] which was generated by IBM *QUEST* data generator [20].

4.2 Accuracy of prediction

Each dataset is read in memory, and then is split into a training set and a testing set using the k-fold cross validation. Once a predictor has been trained, each sequence of the testing set is split into two parts, the *query prefix* and the *query suffix*. The size of each can be defined through a parameter in advance. Then a trained prediction model is called to give answers for every prefix in the testing set. A prediction answer for a query prefix is accurate if it appears within the query suffix[7]. The accuracy rate is the ratio of accurate predictions to the total number of test sequences. Each prediction model has been trained and tested using k-fold cross validation with k = 14 to obtain a low variance for each run.

Accuracy results are shown in Table 1. Our prediction model provides better accuracy than any other lossy predictor for SIGN, KOSARAK and FIFA datasets. At the same time, we can observe that SuBSeq has an overall better accuracy than any predictor for MSNBC and BIBLE_CHAR. However, if we take into consideration the accuracy variation of CPT+ (as show in the Table 1

[5] Available at `http://www.philippe-fournier-viger.com/spmf`
[6] Details about QUEST exported data, are available at `github.com/rafkt/SUBSEQ`
[7] Same evaluation approach was followed for CPT+[6]

at CPT+ column in a [min-max] range) based on its different possible parameter tunes, then SuBSeq provides an overall better accuracy performance for KOSARAK and FIFA as well. Thus, CPT+ gets less competitive if it is not finely tuned making SuBSeq more attractive.

Datasets	**DG**	**TDAG**	**CPT+**	**subSeq**	**Mark1 (PPM)**	**AKOM**	**LZ78**	**SPiCe baseline**
BMS	**36**	7	[30-**38**]	33	30	31	33	0.19
SIGN	2	0	**[26-34]**	**23**	4	7	5	4
MSNBC	55	31	[49-**59**]	**64**	38	48	43	30
BIBLE_WORD	6	23	[0 - 22]	**29**	11	**32**	18	2
BIBLE_CHAR	3	79	[1 - 80]	**88**	16	**81**	65	6
KOSARAK	30	1	**[31-37]**	**34**	23	20	20	0.6
FIFA	25	7	[18 - **34**]	**29**	23	26	25	0.38

Table 1: Prediction models and their accuracy in %. First and second best performers are in bold

4.3 Performance

The Memory of SuBSeq was measured by using the relevant api in sdsl library. The memory for the rest of the predictors was measured through IPredict. We compared the different prediction models through the ratio of their memory usage over the training set binary size. In the Table 2, SuBSeq is the most consistent and most memory efficient prediction model. It uses an average memory of up to 2.2 times the memory of the input training set binary size. Prediction models like TDAG and CPT+ appear to be highly inconsistent. TDAG can utilise space between 70 to 2500 times the input binary size while CPT+ between 0.5 to 80 times; indicating an unpredictable performance.

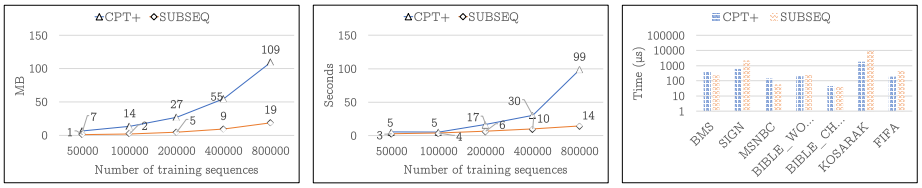
The running time of SuBSeq was directly compared to CPT+ for various datasets (Figure 3c) in respect of the testing-phase (and training-phase). Evaluations also included input data of an increasing σ, n, d using the QUEST generator. The results showed competitive and consistent performance for SuBSeq in comparison to CPT+.

4.4 Optimisation discussion

Our current implementation of SuBSeq is not fully optimised yet. Experimental evaluation showed that 90% of the time needed from SuBSeq to answer a query, it is spent for neighbour expansion. Further experiments revealed that

Datasets	DG	TDAG	CPT+	CPT	subSeq	Mark1 (PPM)	AKOM	LZ78
BMS	4.87	136.34	9.01	15.58	2.14	1.63	26.05	5.60
SIGN	2.96	124.51	0.54	10.86	1.73	1.69	38.07	5.08
MSNBC	0.06	176.29	3.19	5.42	2.14	0.06	13.71	4.14
BIBLE_WORD	6.07	77.72	11.10	12.74	1.90	1.70	20.83	3.40
BIBLE_CHAR	0.68	2689.15	3.38	6.46	2.18	0.25	51.69	42.77
KOSARAK	6.76	126.92	81.49	86.43	1.67	21.17	30.62	4.86
FIFA	2.98	90.74	4.88	6.64	1.60	1.15	23.40	3.59

Table 2: Ratio of prediction model memory to training binary size ($M * \lceil \log(\sigma) \rceil$)



(a) Memory (b) Time (c) various datasets

Fig. 3: Testing time performance of CPT+ and SuBSeq

in average only a 45% of the executed rank operations are unique per query. Thus, preventing neighbour expansion from performing excessive rank calls in the wavelet tree, would optimise the speed performance of SuBSeq for datasets with large σ. Figure 3c shows that for a dataset like KOSARAK ($\sigma = 654,987$), SuBSeq performance is less competitive. One way to minimise excessive rank calls is to store (retrieve) each rank result in (from) a trie-based data structure.

5 Conclusion

Lossless sequence predictors are often very accurate but can consume a large amount of memory. To address this issue, this paper presented a novel predictor named SuBSeq that is lossless and utilizes the succinct Wavelet Tree data structure and the Burrows-Wheeler Transform to compactly store and efficiently access training sequences for prediction. Experimental results have shown that SuBSeq has a very low and predictable memory consumption (varying 1.6 to 2.2 times the binary size of D) and excellent accuracy in comparison to state-of-the-art predictors on real datasets. Moreover, SuBSeq is mostly parameter-free. Future work includes optimising SuBSeq neighbour expansion along with its overall speed performance.

References

1. Balle, B., Eyraud, R., Luque, F.M., Quattoni, A., Verwer, S.: Results of the Sequence PredIction ChallengE (SPiCe): a Competition on Learning the Next Symbol in a Sequence. In: Proc. 13th International Conference in Grammatical Inference. vol. 57. JMLR W&CP, Delft, Netherlands (2016)
2. Burrows, M., Wheeler, D.: A block-sorting lossless data compression algorithm. Tech. Rep. 124, Digital Equiptment Corporation (1994)
3. Cleary, J., Witten, I.: Data compression using adaptive coding and partial string matching. IEEE Trans. Commun. 32(4), 396–402 (1984)
4. Gog, S.: simongog/sdsl-lite (2015), `https://github.com/simongog/sdsl-lite`
5. Gog, S., Petri, M.: Optimized succinct data structures for massive data. Software, Practice and Experience 44(11), 1287–1314 (2014)
6. Gueniche, T., Fournier-Viger, P., Raman, R., Tseng, V.S.: Cpt+: Decreasing the time/space complexity of the compact prediction tree. In: Proc. PAKDD. pp. 625–636 (2015)
7. Gueniche, T., Fournier-Viger, P., Tseng, V.S.: Compact prediction tree: A lossless model for accurate sequence prediction. Advanced Data Mining and Applications pp. 177–188 (2013)
8. Kärkkäinen, J., Kempa, D., Puglisi, S.J.: Hybrid compression of bitvectors for the FM-index. In: Proc. DCC. pp. 302–311. IEEE (2014)
9. Laird, P., Saul, R.: Discrete sequence prediction and its applications. Machine Learning 15(1), 43–68 (Apr 1994)
10. Manber, U., Myers, E.W.: Suffix arrays: A new method for on-line string searches. SIAM J. Comput. 22(5), 935–948 (1993)
11. Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)
12. Navarro, G.: Wavelet trees for all. Journal of Discrete Algorithms 25, 2–20 (2014)
13. Navarro, G., Mäkinen, V.: Compressed full-text indexes. ACM Computing Surveys 39(1), article 2 (2007)
14. Okanohara, D., Sadakane, K.: Practical entropy-compressed rank/select dictionary. In: Proc. ALENEX. pp. 60–70. SIAM (2007)
15. Padmanabhan, V.N., Mogul, J.C.: Using predictive prefetching to improve world wide web latency. SIGCOMM Comput. Commun. Rev. 26(3), 22–36 (Jul 1996)
16. Pitkow, J., Pirolli, P.: Mining longest repeating subsequences to predict world wide web surfing. In: USITS'99 Proceedings of the 2nd conference on USENIX Symposium on Internet Technologies and Systems - Volume 2. pp. 139–150 (1999)
17. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
18. Rjeily, C.B., Badr, G., Al Hassani, A.H., Andres, E.: Predicting heart failure class using a sequence prediction algorithm. In: 2017 Fourth International Conference on Advances in Biomedical Engineering (ICABME). pp. 1–4. IEEE (2017)
19. Tax, N.: Human activity prediction in smart home environments with lstm neural networks. In: 2018 14th International Conference on Intelligent Environments (IE). pp. 40–47. IEEE (2018)
20. Zheng, Z., Kohavi, R., Mason, L.: Real world performance of association rule algorithms. In: Proc. ACM SIGKDD. pp. 401–406. ACM (2001)
21. Ziv, J., Lempel, A.: Compression of individual sequences via variable-rate coding. IEEE Trans. Inf. Theor. 24(5), 530–536 (1978)