Text Prediction and Classification Using String Matching

Byron Knoll
Department of Computer Science
University of British Columbia

Abstract

This paper introduces a simple dynamic programming algorithm for performing text prediction. The algorithm is based on the Knuth-Morris-Pratt string matching algorithm. It is well established that there is a close relationship between the tasks of prediction, compression, and classification. A compression technique called Prediction by Partial Matching (PPM) is very similar to the algorithm introduced in this paper. However, most variants of PPM have a higher space complexity and are significantly more difficult to implement. The algorithm is evaluated on a text classification task and outperforms several existing classification techniques.

1 Introduction

This paper introduces a simple dynamic programming algorithm for performing text prediction. The algorithm is very similar to a compression technique called Prediction by Partial Matching (PPM) (Cleary and Witten, 1984). PPM is one of the best lossless text compression techniques, so has been subject to a substantial amount of research. It consistently performs well on data compression benchmarks. There are a large variety of PPM implementations. However, the specific variation of PPM discussed in this paper does not appear to have been previously published. The algorithm is based on the Knuth-Morris-Pratt (KMP) string matching algorithm (Knuth, Morris, and Pratt, 1977).

The remainder of this paper is organized into five main sections. Section 2 provides background on the text prediction problem and its relationship to data compression and classification. Section 3 gives an overview of PPM and KMP, two algorithms which are closely related to this work. Section 4 provides a high level description of the algorithm. Section 5 evaluates the algorithm on prediction and classification tasks. Finally, section 6 discusses the results and speculates at possible future work. The appendix provides a Java implementation of the algorithm introduced in this paper.

2 Background

Text prediction can be considered as a sequential process over time with an input stream of characters. The task is to predict the next character given a string representing the input history. In this paper the first character of a string represents oldest input and the last character represents the newest input. For example, given the string "abababa" a good guess for the next character would be 'b' since 'b' always follows 'a' in the input history. It is well established that there is a close relationship between the tasks of prediction, compression, and classification (Marton, Wu, and Hellerstein, 2005). An algorithm which is good at text prediction will also be good at text classification and text compression. The task of text prediction is not necessarily limited to natural language. For example, an alphabet of two letters can be used for any binary file, regardless of the type of data it contains.

Text prediction can be used for a variety of applications. For example, it can be used to minimize the number of keystrokes required to type a given text (Garay-Victoria and Abascal, 2006). This can be especially useful for slow input methods such as mobile phone keypads. In addition, it can help increase the communication rate for people with disabilities. Another use for text prediction is for denoising data. Character predictions can actually be more valid than the actual input in certain scenarios, such as the case of spelling mistakes. Spelling mistakes can be considered as "noise" in the data which can be corrected using a predictive filter. Finally, another example of a common application for text prediction is the automated completion of search terms used by several search engines.

Classification and compression also have fairly obvious applications. Considering the three tasks of prediction, classification, and compression together covers a large range of problems encountered in the fields of artificial intelligence and machine learning. The relationship between these three tasks is explored in more detail below.

2.1 Relationship Between Prediction and Compression

Text prediction algorithms can assign a probability distribution to characters in an alphabet corresponding to the probability of each character being next in the input stream. This probability distribution can be combined with a coding scheme such as arithmetic coding or Huffman coding to compress data. In fact, a measurement called cross entropy can be used to estimate the average number of bits needed to code the data. For a sequence of N characters x_i, and a probability $p(x_i)$ assigned to each character by the prediction algorithm, the cross entropy can be defined as:

$$-\sum_{i=1}^{N} \frac{1}{N} log_2 p(x_i)$$

This gives the expected number of bits needed to code the string. Another common metric used to compare text prediction algorithms is perplexity, which can be defined as two to the power of cross entropy:

$$2^{-\sum_{i=1}^{N} \frac{1}{N} log_2 p(x_i)}$$

In 1991, a trigram model was used on a large corpus of one million English words to achieve a perplexity score of 247 per word, corresponding to a cross entropy of 7.95 bits per word or 1.75 bits per letter (Brown, Della Pietra, Della Pietra, Lai, Mercer, 1992). On this corpus, ASCII coding has a cross entropy of 8 bits per character, Huffman coding has 4.46, and the UNIX command `compress` has 4.43. On more specialized corpora it is possible to achieve lower perplexity scores than for more general corpora. Recently, a word perplexity score of 96.9 was reported on the Associated Press corpus using a technique called stochastic memoization (Wood, Archambeau, Gasthaus, James, and Teh, 2009). This is significantly lower than the perplexity scores reported for competing approaches.

2.2 Relationship Between Prediction and Classification

Classification is a task in which items must be categorized into groups based on a training set of previously labelled items. Any prediction or compression algorithm can be used for classification. This can be done by first separating the training data into categories based on their labels. When unlabelled data needs to be classified into a category, each training category can be used as a separate training set for the prediction/classification algorithm. In the case of prediction, the prediction error for the data is compared using each category as a training set. The data can be classified as being in the category which results in the lowest prediction error. Similarly, in the case of compression the file size of the data is compared when compressing it using each training category. The data can be classified as being in the category which results in the lowest compressed file size.

Consider a concrete example of binary classification. Suppose there are a set of documents which have been labelled as being either funny or unfunny. These training documents can be separated into the two categories. Given a new unlabelled document, the goal is to classify it as either being funny or unfunny. One approach to doing this is using a text prediction algorithm. The prediction error of the document can be tested using the funny training data by first inputting all the funny training data as a string to the prediction algorithm. The prediction error of the document can be calculated from the number of incorrect character predictions made when sequentially inputting the document's text. Similarly, the prediction error of the document using the unfunny training set can be computed. Finally, the document can be classified as being in the category which results in the lowest prediction error. Another classification approach is using a data compression algorithm. First, the file size of the funny training data compressed alone can be compared to the file size of the document appended to the funny training data. Subtracting the two sizes results in the amount of data needed to code the document (using the funny training set). Similarly, the amount of data needed to code the document using the unfunny training set can be computed. The document can be classified as being in the category which results in the smallest amount of data needed to code it.

Given the choice between classification using a text prediction algorithm and the same algorithm used for compression, there is a practical advantage to using prediction error instead of compressed file size. Using prediction error avoids the computational overhead involved when performing compression. This overhead includes using a coding scheme (such as arithmetic coding) and writing compressed files to disk, which can be a very slow operation.

Figure 1 summarizes the directed relationships between prediction, classification, and compression which have been discussed in this paper. That is, any prediction algorithm can be used for compression. Additionally, any prediction or compression algorithm can be used for classification. An argument might be made for the bidi-

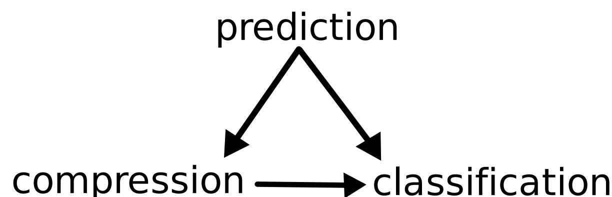

Figure 1: Directed relationships between prediction, compression, and classification.

rectionality of any of these relationships. However, the relationships presented in this paper seem to be the most intuitive.

3 Related Work

3.1 Knuth-Morris-Pratt Algorithm

The naïve approach to matching a string S of length M to a text T of length N has a time complexity of $O(M \times N)$. This approach involves simply iterating through N positions of T and for each position checking whether the next M characters match S. The KMP algorithm decreases this time complexity to $O(M + N)$. There are two phases of the KMP algorithm. The first involves iterating through the M characters of S and building a table of size M. The second involves iterating through the N characters of T and finding matches.

An intuitive understanding of the KMP algorithm can be gained by considering a simple example. Suppose we are trying to match a string S = "abcabz" to the text T = "abcabcabz". Iterating through T, the first five characters match exactly to S. However, as soon as we reach the sixth character there is a mismatch between the 'z' in S and 'c' in T. In the naïve approach this mismatch would force us to return to the second index of T and try matching it to the beginning of S. However, the observation can be made that when we found the mismatch at position six, we already matched "ab" at positions four and five. These happen to be the first two characters of S. This means we can just continue trying to match position six of T with position three of S.

The purpose of building the table for S is that it acts as a failure function for when we encounter a mismatched character. It contains an index in S which allows us to continue matching the original text T without backtracking. This table can be constructed in $O(M)$ time. After the table is constructed, it takes $O(N)$ time to iterate through T.

3.2 Prediction by Partial Matching

Although there are many variants of the PPM algorithm, they all share a common concept. The idea is that a good way to make a prediction about the next character in a sequence is to try to match the sequence to some part of the input history and make the prediction based on what character comes next in the history. For example, consider the string "abczabczabc". A good guess for the next character in the sequence would be 'z' since 'z' always comes after 'c' in the history. In addition, 'z' always comes after the string "bc". Furthermore, 'z' always comes after the string "abc". It should be clear that making longer matches is preferable since they are less likely to occur by chance. Under the assumption that patterns exist in the input stream, longer matches will lead to better predictions. This is essentially a task of temporal pattern recognition.

Matching the recent input sequence to the input history can be represented as a string matching problem. The task is to find the longest matching string between recent input and the history string. When the longest match is found, a prediction can be returned as the character which occurs immediately after the match in the history string.

This model is actually equivalent to the use of n-grams. In fact, n-grams are actually (n-1) order Markov models. The length of the match made between the recent input sequence and the history string determines the order of the Markov model. This means that in PPM the order of the model is adaptively changed based on the length of matches occurring in the input string. If no matches of a particular length occur in the string, the order of the model must be reduced.

Most PPM implementations use a fixed maximum size for the order of the Markov model. This is done to reduce the time and space complexity of the algorithm. The majority of PPM implementations use exponential memory in relation to the maximum length of the Markov model. In 1997 a variant of PPM was introduced called PPM* which uses an unbounded order Markov model (Cleary and Witten, 1997). The stochastic memoizer mentioned in section 2.1 also uses an unbounded order Markov model to achieve record breaking perplexity scores.

4 Algorithm Description

The algorithm introduced in this paper uses an unbounded order Markov model. In addition, it uses linear memory which is an improvement over the exponential memory needed by most PPM implementations. However, for predicting each new character of an input stream it has a time complexity of $O(N)$ where N is the history size. This means that for a given file of length N, the time complexity to predict every byte of that file is $O(N^2)$. The time complexity for most PPM variations range between $O(N)$ and $O(N^2)$. Time complexities below $O(N^2)$ can be achieved using suffix tries.

The appendix of this paper has a full Java implementation of the algorithm. Of particular note is the simplicity

of the algorithm in comparison to other PPM implementations. If the amount of data to be processed is relatively small so that the $O(N^2)$ time complexity is not a concern, this algorithm could be preferable to the use of other PPM implementations. This is due to its linear memory usage, unbounded Markov order, and simple implementation.

The algorithm relies upon KMP string matching. Consider the input string "zabracadabra". Reversing the string results in another string S = "arbadacarbaz". Now consider matching S to the text T = "rbadacarbaz" (the same as S except missing the first character). Running KMP string matching on S and T will result in a sequence of character mismatch events. Each mismatch has a corresponding match length indicating how much of S matches T. The mismatches (S:T:length triples) for this example are a:r:0, a:b:0, r:d:1, r:c:1, and d:z:4. These mismatches indicate that the longest match in the string is four. Given the longest match of "abra", a prediction of 'c' can be made as the next character.

In the Java implementation provided in the appendix, a map data structure is used to store information about the longest matches. If there are multiple matches of the same length, a prediction for the next character can be made by using the most frequent character prediction among the matches. The stored matches can also be used to optimize the speed of future predictions. If a new character was correctly predicted by one of the matches stored in the data structure, the longest matches do not need to be recomputed. This allows a prediction to be returned in $O(1)$ time. However, if the new character was not predicted by one of the stored matches, new matches will need to be recomputed from the entire history in $O(N)$ time. Therefore, this algorithm will run faster when it is good at predicting the input data.

5 Evaluation

Since PPM has already been extensively studied for the task of compression, this paper will focus on evaluating the algorithm on prediction and classification tasks.

5.1 Prediction

The majority of evaluation on text prediction algorithms is done by comparing perplexity metrics. Since the algorithm introduced in this paper simply outputs the most likely character instead of assigning a probability distribution to all the characters, the perplexity metric cannot be used. It should be noted that it is not necessarily difficult to assign a probability distribution to the characters but this was not discussed in this paper to avoid additional complexity. Instead, the error rate of character predictions on various data can be reported. The Calgary corpus is a popular dataset to compare the performance of compression algorithms. Unfortunately, most publications about this corpus report perplexity scores and compressed file sizes instead of error rates for character prediction. However, examining the error rates for files in the Calgary corpus is still a useful exercise because it allows comparison of error between different types of data. In addition, the error scores reported in this paper can be used as a comparison metric for future work.

File	Size (KiB)	Description
bib	111.261	structured text (bibliography)
book1	768.771	text, novel
book2	610.856	formatted text, scientific
geo	102.400	geophysical data
news	377.109	formatted text, script with news
obj1	21.504	executable machine code
obj2	246.814	executable machine code
paper1	53.161	formatted text, scientific
paper2	82.199	formatted text, scientific
pic	513.216	image data (black and white)
progc	39.611	source code
progl	71.646	source code
progp	49.379	source code
trans	93.695	transcript terminal data

Table 1: File size and description of Calgary corpus files.

Table 1 provides a description of the different files contained in the Calgary corpus. Table 2 provides the average byte prediction error for files in the Calgary corpus. Comparing byte prediction error instead of binary error or some other granularity was chosen purely for implementation convenience. Of course, choosing a smaller granularity such as binary prediction error results in lower error rates, but should preserve the same relative performance between different types of data. The 'pic' file stands out as having an extremely good prediction rate. This is likely due to the fact that it is not compressed so has a lot of redundant information in comparison to its underlying Kolmogorov complexity. It is also interesting to compare the error rates between similar types of data. For example, the 'book1' novel has a much higher error rate than the scientific text 'book2'.

Figure 2 shows how the average prediction error rate for a novel changes sequentially from the beginning of the text to the end. The novel is in ASCII format and was obtained from Project Gutenberg (http://www.gutenberg.org). The prediction error at the end of the text was 41.066%. The graph indicates that the rate of change of error decreases over time. One use for this type of information is that it can help provide an estimate of the maximum number of bytes that are needed for decreasing the error rate. For example, if there is no significant drop in error rate after 1MiB of input history, then the history size can be limited to 1MiB to help increase computational performance of the algorithm. By

File	Error Percentage
bib	32.768%
book1	47.149%
book2	37.032%
geo	64.346%
news	39.693%
obj1	45.999%
obj2	33.513&
paper1	39.916%
paper2	43.023%
pic	11.289%
progc	37.713%
progl	26.928%
progp	24.066%
trans	20.476%

Table 2: Average byte prediction error on Calgary corpus files.

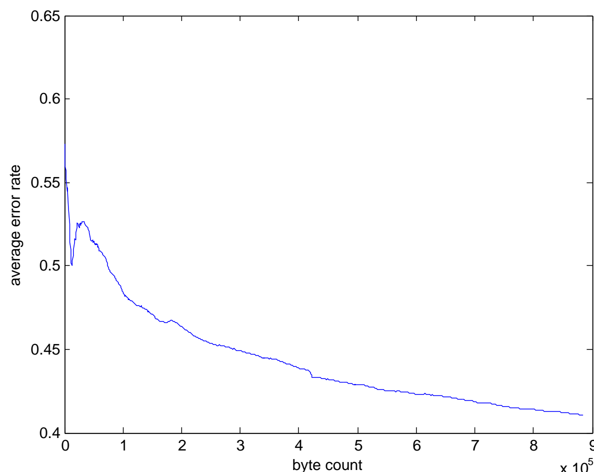

Figure 2: Average error rate over time for each byte of the novel Twenty Thousand Leagues Under the Sea (Jules Verne, 1870).

limiting the history size, the computational complexity of the algorithm is reduced from $O(N^2)$ to $O(N)$.

5.2 Classification

The dataset chosen for this classification task is from an undergraduate machine learning course at the University of British Columbia. The dataset was used in a class competition to give bonus marks to the students with the lowest test error. Since a substantial amount of bonus marks were available to students for performing well in the contest, the incentive for students to invest a lot of time/effort in the competition was significantly increased. The task was to classify Wikipedia articles as either being part of a particular category or not part of it. In the training set the articles were labelled as either positive (part of the category) or negative (not part of the category). For example, if the category was "hobbies" then all of the positive training articles belong to the hobbies category and none of the negative training articles belong to the hobbies category. The negative training articles were not necessarily part of the same category. The actual category was not given. There were 100 positive training articles, 100 negative training articles, 100 positive test articles, and 100 negative test articles.

Method	Error Percentage
PRED ORIG	12%
PRED LOW	10.5%
PRED STEM	10%
PAQ ORIG	11%
PAQ LOW	16%
PAQ STEM	23.5%

Table 3: Classification error on the test set (200 articles). Three different training sets were used: original ASCII text (ORIG), text converted to lowercase characters (LOW), and text converted to word stems (STEM).

The results from six classification methods are summarized in table 3 and table 4. The technique used to do the classification was the same as that described in section 2.2. PRED refers to the text prediction algorithm described in section 4 and PAQ refers to the PAQ8L data compression algorithm. The PAQ data compression algorithm was chosen because it has top rankings on several benchmarks measuring compression ratio. Performing preprocessing on the training set had a significant effect on the error rates. The two preprocessing steps used were converting all characters to lowercase (LOW) and performing word stemming (STEM) on the lowercase letters. Word stemming was done using the Porter algorithm (Porter, 1980). Examining table 4 indicates that several of the methods were significantly better at classifying documents in the POS set than the NEG set. This bias is especially noticeable for the PAQ STEM method. The bias does not appear to be present in PAQ ORIG. It is unclear why the PAQ algorithm performed worse when the preprocessing steps were performed. In the case of PRED, it is expected that the preprocessing steps should decrease the error because they allow longer matches to be discovered.

Table 5 provides the error percentages of the top six participants in the class competition. A variety of different classification approaches were used including neural networks, decision trees, n-gram based methods, and support vector machines. It should be noted that all of these techniques ran significantly faster than any of the PRED or PAQ methods. All six of the methods in table 5 ran in the order of a few minutes, while the six PRED/PAQ methods took several hours to run. Slower

Method	Wrong in POS	Wrong in NEG
PRED ORIG	7	17
PRED LOW	5	16
PRED STEM	5	15
PAQ ORIG	11	11
PAQ LOW	3	29
PAQ STEM	3	44

Table 4: Number of errors in the positive test set (POS) and negative test set (NEG). There are 100 articles in each test set.

Name	Error Percentage
Fisher LD Hill Climb & NBayes	9%
NaiveB	12.5%
turtle_star	12.5%
Classifoo	12.5%
boosted?_perceptron	13%
BetterLateThanNever	13%

Table 5: Top six results from classification competition. There were a total of 18 entries.

runtime performance is one of the disadvantages to using compression techniques with a high compression ratio. When comparing table 3 to table 5 we can see that all three of the PRED methods beat 17 out of the 18 entries in the class contest.

6 Discussion and Future Work

The classification results in section 5.2 seem very promising. Compared to the majority of the entries in the class competition, the approach used in this paper is very easy to implement. Coding complexity could conceivably be an important factor for the development of certain applications under time pressure. In addition, the lack of any parameters to tune also decreases the amount of time needed to deploy the code. However, the high time complexity of the algorithm may make it unsuitable for certain large datasets.

The performance of several other approaches were evaluated before choosing to focus on PPM in this paper. These approaches included linear predictive coding, temporal neural networks, and hierarchical temporal memory (HTM) (Hawkins and Blakeslee, 2004). For the task of binary prediction on the Wikipedia dataset used in section 5.2, linear predictive coding had an error rate of 32%, hierarchical temporal memory 26%, and temporal neural networks 28%. In comparison, the algorithm in this paper had an error rate of 11%.

Two modifications to the algorithm in this paper were explored. One modification involved using approximate string matching instead of exact string matching. The motivation for this idea was that approximate string matching allows for longer matches which could potentially decrease the error rate of predictions. A dynamic programming algorithm was implemented for approximate string matching using the minimum Levenshtein distance. This algorithm was significantly more computationally expensive than the approach used in this paper and appeared to have a significantly higher error rate. Based on these results, this approach was abandoned.

Another modification to the algorithm explored was the use of ensemble voting to make predictions. The ensemble voting was performed between different orders of Markov models (different match lengths). Longer matches were given a higher vote and shorter matches were given a smaller vote. Imagine if there is only one match of length 50 and 100 matches of length 49. The matches of length 49 are likely to contain some predictive value, so it makes intuitive sense to give them some weight. Several weighting functions were experimented with. Overall, the results of the ensemble voting seem to be slightly better than the algorithm presented in this paper. However, it was not clear whether this difference was statistically significant. This is a promising area for future work. Assigning weights to the different order Markov models also simplifies the task of creating a probability distribution over the characters. However, the task of assigning good weights is a difficult problem and remains an active area of research.

One approach to assigning the weights is to base them upon the empirical prediction accuracy of the different match lengths. For example, if the longest match correctly predicts the next character 42% of the time, 0.42 would be a good weight assigned to the character predicted by the longest match. Since a given match length can predict multiple characters, if the second most likely character predicted by the longest match is correct 5% of the time, a corresponding probability of 0.05 can be assigned to the second most likely character of the longest match. Similarly this can be done for lower match lengths.

Another potential area for future work is limiting the history size. The results in figure 2 and corresponding discussion in section 5.1 indicate that a limited history size may not necessarily have a significant impact on prediction accuracy. One approach to limiting the history size is to simply use a sliding window and forget everything before a certain point in history. However, this naïve approach can be improved upon. Ideally, only portions of the history which will never be matched should be removed. However, it is impossible to determine whether a particular portion of the history may be matched at some point in the future. If we keep track of statistics on how often different characters in the history string are matched, this might provide some indication of how likely those characters will be matched in the future.

This statistic will also need to be weighted by how recently the character occurred, since recent sections of the history have less opportunity to be matched compared to older sections of the history. This statistic may be used as a heuristic for determining which sections of the history can be forgotten.

Certain properties of PPM can be compared to how the human brain operates. It is clear that the brain stores memories of the past and that these memories can be retrieved based on their similarity to recent events. This is exactly the same principle that PPM operates on. There is also evidence that the task of prediction plays a fundamental role in human intelligence and behaviour (Hawkins and Blakeslee, 2004). However, there are clearly differences in the capabilities of the human brain when compared to PPM.

One remarkable property of the brain is its massive parallelism in information processing. In contrast, PPM works on a single sequential character stream. Another difference is that the brain can perform higher level functions which require accessing memories in a non-sequential order. That means that certain predictions may be a function of several non-contiguous segments of the input history. For example, consider the task of adding "46+54". Although this particular string sequence may never have occurred in a person's input history, the individual components of numbers and the addition operator may have been encountered at different points in time. In order to predict what the answer of this operation is, a non-sequential function of memory access is required. It is possible that by stacking PPM predictors on top of each other, higher level patterns can be recognized. HTM provides a framework which may help parallelize and stack individual PPM predictors. In fact, PPM performs a function very similar to what is required by nodes in HTM. This would be another interesting area for future research.

References

Brown, P., Della Pietra, S., Della Pietra, V., Lai, J., and Mercer, R. 1992. *An estimate of an upper bound for the entropy of English*. Computational Linguistics, 18(1), pp 31-40.

Cleary, J., and Teahan, W. 1997. *Unbounded length contexts for PPM*. Comput. J. 40, 2/3, pp 6775.

Cleary, J., and Witten, I. 1984. *Data compression using adaptive coding and partial string matching*. IEEE Transactions on Communications, Vol. 32 (4), pp 396-402.

Garay-Victoria, N., and Abascal, J. 2006. *Text prediction systems: A survey*. Univ. Access. Inf. Soc. 4, pp 188-203.

Hawkins, J., and Blakeslee, S. 2004. *On Intelligence*. New York: Holt.

Knuth, D., Morris, J., and Pratt, V. 1977. *Fast Pattern Matching in Strings*. In SIAM Journal on Computing, pp 323-350.

Marton, Y., Wu, N., and Hellerstein, L. 2005. *On compression-based text classification*. In Proceedings of the European Colloquium on IR Research (ECIR), pp 300-314.

Porter, M. 1980. *An algorithm for suffix stripping*. Program, 14(3) pp 130-137.

Wood, F., Archambeau, C., Gasthaus, J., James, L., and Teh, Y.W. 2009. *A stochastic memoizer for sequence data*. In ICML 09: Proceedings of the 26th Annual International Conference on Machine Learning, pp 1129-1136.

7 Appendix: Java Source Code

```
import java.util.Iterator;
import java.util.LinkedList;
import java.util.TreeMap;

public class Predictor {
    private TreeMap<Character, LinkedList<Integer>> tree = new TreeMap<Character, LinkedList<Integer>>();
    private int longestMatch = -1;

    /**
     * @return predicted next character of the string
     */
    public char predict(String str) {
        if (str.length() == 0)
            return '0';
        if (str.length() == 1) {
            longestMatch = 0;
            return str.charAt(0);
        }
        if (tree.containsKey(str.charAt(str.length() - 1))) {
            longestMatch++;
            LinkedList<Integer> pred = tree.get(str.charAt(str.length() - 1));
            tree.clear();
            for (int pos : pred) {
                char c = str.charAt(pos + 1);
                LinkedList<Integer> temp;
                if (tree.containsKey(c))
                    temp = tree.get(c);
                else
                    temp = new LinkedList<Integer>();
                temp.add(pos + 1);
                tree.put(c, temp);
            }
        } else {
            longestMatch = -1;
            int m = 1;
            int i = 0;
            int[] table = createTable(str);
            while (m + i < str.length()) {
                if (str.charAt(str.length() - i - 1) == str.charAt(str.length() - (m + i) - 1))
                    i++;
                else {
                    insertPrediction(str.charAt(str.length() - m), i, str.length() - m);
                    m = m + i - table[i];
                    if (table[i] >= 0)
                        i = table[i];
                }
            }
            if (i > 0)
                insertPrediction(str.charAt(str.length() - m), i, str.length() - m);
        }
        char prediction = '0';
        int maxCount = 0;
        Iterator<Character> it = tree.keySet().iterator();
        while (it.hasNext()) {
            char key = it.next();
            int count = tree.get(key).size();
            if (count > maxCount) {
                prediction = key;
                maxCount = count;
            }
        }
        return prediction;
    }

    private void insertPrediction(char c, int i, int pos) {
        if (i > longestMatch) {
            tree.clear();
            longestMatch = i;
        }
        if (i < longestMatch)
            return;
        LinkedList<Integer> pred = null;
        if (tree.containsKey(c))
            pred = tree.get(c);
        else
            pred = new LinkedList<Integer>();
        pred.add(pos);
        tree.put(c, pred);
    }
```

```
    private int[] createTable(String str) {
        int pos = 2;
        int cnd = 0;
        int[] table = new int[str.length() - 1];
        table[0] = -1;
        while (pos < str.length() - 1) {
            if (str.charAt(str.length() - pos) == str.charAt(str.length() - cnd - 1)) {
                table[pos] = cnd + 1;
                pos++;
                cnd++;
            } else if (cnd > 0)
                cnd = table[cnd];
            else {
                table[pos] = 0;
                pos++;
            }
        }
        return table;
    }
}
```