The Fitness of Context Tree Prediction under different Context-Ranking Estimators

James Halliday

1 Abstract

This paper explores the rate at which a variable order markov model built with a context-tree weighting method is capable of predicting to the end of some finite binary sequence using experimental results. The model is run against different test sequences with four estimators derived from statistical tests or discussed in the literature. It is the aim of these experiments to provide insights into the construction of a real-time streaming prediction framework for arbitrary binary sequences. Krichevsky-Trofimov and symmetric confidence interval-derived estimators proved to be comparably effective compared to variance and depth.

2 Introduction

There are a great variety of options available when shopping for general-purpose predictors which operate on arbitrary binary sequences, each of which makes its own set of assumptions and trade-offs. In cases where much is already known about the order and distribution of the sequences in question, a simple fixed-order markov model will often suffice.

For more complex applications, hierarchical, layered, or hidden markov models can be used, but require substantial amounts of prior information or else require a large amount of computation and memory. When little is known about the incoming data stream, such as with a general-purpose compression algorithm, few assumptions are available. Variable order markov models are advantageous under these considerations and provide a basis for well-known compression algorithms such as Lempel-Ziv-Welch.

Variable order markov models (VOMs) make use of a context of states in order make predictions about future states. For instance, in the sequence "abac", the context "aba" might map to the state "c" or likewise with "ab"

mapping to "ac". The number of these associations is z^n, where z is the number of symbols in the alphabet of the model and n is the length of the model. In order to combat this exponential growth, VOMs often see fit to overlook or discard certain associations, enforcing a maximum order or a maximum data size for the resulting tree.

3 Building Context Trees

The context tree weighting method (CTWM) is used to construct a tree of unbounded size for the purposes of this experiment. The CTWM has been shown experimentally to outperform other classes of VOMs on highly redundant sequences such as English text and performs well on highly irreducible sequences such as protein genomes too. [1] The CTWM builds its tree by remembering the last active contexts represented as nodes and propagating these activity levels up the tree in the corresponding direction for each bit in the input sequence: left for zero and right for one. A count is maintained at each node which is incremented every time the node is activated. Predictions can be made at any time on this structure by ranking the active states with an estimator which takes the counts of the immediate descendants as parameters. For the purposes of simplicity, this model operates on a binary alphabet.

This document, meanwhile, is a literate haskell program which generates typeset TeX documents and can also be run directly to verify the results of this experiment. Being a program, this document will now import some important libraries of great import:

```
module Main where

import Data.Bits
import Data.List.Split (splitEvery)
import Data.Ord (comparing)
import Maybe
import Data.List (elemIndex, tails, maximumBy, sortBy, intersperse, isPrefixOf)
import Data.Function (on)
import Data.Number.Erf (normcdf)
```

A Tree data type is defined recursively with a Root and Nodes, both of which may have children which are Trees themselves. Usage statistics are included and updated on the Root node for the curious. Nodes also have an

activity level, a digit representing their contribution to the context, and a count for providing the estimators with something to look at.

```
data Tree =
    Root {
        children :: Children,
        used :: Int
    } |
    Node {
        children :: Children,
        active :: Bool,
        digit :: Bool,
        count :: Int
    } deriving (Eq, Show)
type Children = [Tree] -- a list with [0,2] elements
```

An empty tree is just a rooted document with 1 node in use, itself.

```
empty :: Tree
empty = Root [] 1
```

The growth of trees is handled by a short function which updates the usage record and delegates to a recursive function which propagates activity levels and generates new nodes at active leaves.

```
grow :: Bool -> Tree -> Tree
grow _ Node{} = error "Can only grow the root"
grow bit root = root' { used = used' } where
   (used',root') = growNode bit root

-- grow a node while counting the number of nodes for grow's used record
growNode :: Bool -> Tree -> (Int,Tree)
-- just step deeper into inactive nodes
growNode bit node@Node{ active = False } = (used,node') where
    node' = node { children = grown }
    used = 1 + sum nUsed
    (nUsed,grown) = unzip $ map (growNode bit) $ children node
-- grow active nodes and the root by passing along activity or creating nodes
growNode bit node = (used,node') where
    used = 1 + sum nUsed + (if hasBit then 0 else 1)
    node' = case node of
```

```
        Root{} -> node { children = cx }
        Node{} ->
            node {
                children = cx,
                active = False,
                -- maintain a count of active traversals
                count = succ $ count node
            }
    -- activate and grow the relevant children
    cx = map (activate bit) $ case hasBit of
        True -> grown
        False -> -- node doesn't have a child matching this bit, make one
            Node {
                children = [],
                active = True,
                digit = bit,
                count = 1
            } : grown
    -- whether the node has a child with the current digit
    hasBit = elem bit $ map digit grown
    (nUsed,grown) = unzip $ map (growNode bit) $ children node
```

It's useful to only set activity for particular digits so that an active node's children can be mapped over with this function when the bit is supplied.

```
activate :: Bool -> Tree -> Tree
activate bit node
    | bit == digit node = node { active = True }
    | otherwise = node
```

Generate a list of all nodes in the tree in order to select which ones are active elsewhere.

```
nodes :: Tree -> [Tree]
nodes tree = tree : (concatMap nodes $ children tree)
```

The active nodes are collected and ranked by the given estimator, with the best context returning its best descendant.

```
nextBit :: Sorter -> Tree -> Bool
nextBit sorter tree = bit where
```

```
    bit = case nx of
        [] -> False
        _  -> follow nx
    follow = digit . maximumBy (comparing count) . children . head
    nx = sortBy sorter actives
    actives = [ n | n <- nodes tree, isActive n && children n /= [] ]
    isActive Root{} = False
    isActive node@Node{} = active node
```

The next bit is then incorporated into the new tree as if it came from the input stream to begin with. An infinite and lazy string of predictions is returned.

```
nextBits :: Sorter -> Tree -> [Bool]
nextBits sorter tree = bit : (nextBits sorter $ grow bit tree) where
    bit = nextBit sorter tree
```

4 Experimental Design

The estimators were measured on the number of bits it took for them to accurately predict to the end of the input sequence.

```
-- return how many bits were needed in order to predict to the end of the string
bitsRequired :: Sorter -> [Bool] -> Int
bitsRequired sorter bits
    | isJust index = fromJust index
    | otherwise = length bits
    where
        index = elemIndex True $ zipWith isPrefixOf matches futures
        -- as each bit is added to the tree
        trees = tail $ scanl (flip grow) empty bits
        -- predict the string that follows from the priors
        futures = map (nextBits sorter) trees
        -- need to match these for bit strings
        matches = tail $ tails bits

sorterResults :: [Bool] -> [(String,Int)]
sorterResults bits = [ (name,req sorter) | (name,sorter) <- sorters ] where
    req = flip bitsRequired $ bits
```

There were 5 test sequences. The first four exhibit significant redundancy and the fifth is the irregular sequence resulting from the concatenation of the binary count from 0 to 19.

```
bitPatterns :: [[Bool]]
bitPatterns = [
        -- some regular sequences or else with regular components
        toBits "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa",
        toBits "abababababababacacacacacacacaca",
        toBits "hello world hello world hello w",
        toBits "Initial noise and then abababab",
        -- and an irregular sequence: a binary concatenation from 0 to 19
        [
            False, True, True, False, True, True, True, False, False, True,
            False, True, True, True, False, True, True, True, True, False,
            False, False, True, False, False, True, True, False, True, False,
            True, False, True, True, True, True, False, False, True, True,
            False, True, True, True, True, False, True, True, True, True, True,
            False, False, False, True, False, False, True, True, False, True,
            False, True, False, True, True
        ]
    ]
```

Some convenience functions were written to convert data to and from human-readable forms and binary decompositions.

```
-- convert a string of ascii characters to a list of bits (not so much for utf8)
toBits :: String -> [Bool]
toBits = concatMap toBit where
    toBit :: Char -> [Bool]
    toBit x = map (testBit $ fromEnum x) [0..7]

-- inverse of toBits: convert some bits to a string
toString :: [Bool] -> String
toString = map toChar . splitEvery 8 where
    toChar :: [Bool] -> Char
    toChar xs = toEnum $ foldl setBit 0 $ map fst $ filter snd $ zip [0..7] xs

prettyResults :: String
prettyResults = concat $ intersperse "\n" results where
```

```
    results = concatMap prettyResult bitPatterns

    prettyResult :: [Bool] -> [String]
    prettyResult bits = test : rs where
        test = "Input: \"" ++ toString bits ++ "\""
        rs = [ "    " ++ name ++ " :: " ++ n | (name,n) <- res ]
        res = [ (name, show n) | (name,n) <- sorterResults bits ]

main :: IO ()
main = putStrLn prettyResults
```

5 Ranking Formulas

With the testing framework and data structures from the previous section in place, the influence of the estimators on the context tree prediction can be measured. Certain parameters are negated in order to cause the highest-quality contexts as judged by an estimator to have the highest values.

```
type Sorter = Tree -> Tree -> Ordering

-- sorters for ranking contexts
sorters :: [(String, Sorter)]
sorters = [ (name,comparing f) | (name,f) <- sorters' ] where
    sorters' = [
            ("depth", depth),
            ("kt", kt),
            ("symCI-90%", symCI 0.90),
            ("symCI-95%", symCI 0.95),
            ("symCI-99%", symCI 0.99),
            ("variance", variance)
        ]
```

Depth is included as a predictor for its use in the Prediction by Partial Match Model [1] and for its simplicity. The depth estimator is the depth of the tree rooted as the context.

```
depth :: Tree -> Float
depth node = case length cx of
    0 -> 0
    _ -> 1 + (maximum $ map depth cx)
    where cx = children node
```

The Krichevsky-Trofimov estimator is used for its use in the CTWM [1]. The formulation from Kawabata and Yanagisawa is used for its ability to work with this model. [2]

```
kt :: Tree -> Float
kt node@Node{} = negate $ num / denom where
    num = (factorial $ zeroes - 0.5) * (factorial $ ones - 0.5)
    denom = factorial $ zeroes + ones
    zeroes = fromIntegral $ sum $ map count $ filter (not . digit) cx
    ones = fromIntegral $ sum $ map count $ filter digit cx
    cx = children node
    factorial = fromIntegral . product . enumFromTo 1 . floor
```

An estimator derived from the proportion test of the confidence interval at 95% certainty is included for its use in ranking algorithms and modified to be symmetric to avoid bias.

```
-- first, a helper function...
confidenceInterval :: Int -> Int -> Float -> (Float,Float)
confidenceInterval p n' alpha = (low, high) where
    low = (phat + zterm - zroot) / denom
    high = (phat + zterm + zroot) / denom
    phat = (fromIntegral p) / n
    qhat = 1 - phat
    zterm = z ^ 2 / (2 * n)
    zroot = z * (sqrt $ phat * qhat / n + z ^ 2 / (4 * n ^ 2))
    denom = 1 + z ^ 2 / n
    n = fromIntegral n'
    z = normcdf $ 1 - alpha / 2

-- and then the (symmetric confidence interval)-based metric
symCI :: Float -> Tree -> Float
symCI conf node = negate $ 2 * (abs $ 0.5 - bound) where
    digitCount d = sum $ map count $ filter ((== d) . digit) cx
    cx = children node
    zeroes = digitCount False
    ones = digitCount True
    n = zeroes + ones
    p = max zeroes ones
    bound = fst $ confidenceInterval p n (1 - conf)
```

The binomial variance is included for its simplicity and ability to predict sequences.

```
variance :: Tree -> Float
variance node = negate $ n * p * (1 - p) where
    n = fromIntegral $ sum $ map count cx
    p = (sum xx) / (fromIntegral $ length xx)
    xx = [ fromIntegral $ (count x) * (fromEnum $ digit x) | x <- cx ]
    cx = children node
```

6 Results

The output of the program follows. The numbers following each test are the number of bits it took for the model to predict the remainder of the sequence. The input strings depicted were send in their binary forms to the predictors.

```
Input: "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
    depth :: 9
    kt :: 9
    symCI :: 9
    variance :: 16
Input: "abababababababacacacacacacacaca"
    depth :: 136
    kt :: 136
    symCI :: 136
    variance :: 137
Input: "hello world hello world hello w"
    depth :: 131
    kt :: 101
    symCI :: 101
    variance :: 197
Input: "Initial noise and then abababab"
    depth :: 210
    kt :: 210
    symCI :: 210
    variance :: 225
Input: "vGVGV"
    depth :: 50
    kt :: 50
```

```
symCI :: 52
variance :: 65
```

Krichevsky-Trofimov converged surprisingly closely with the symmetric confidence interval, with only a 2-bit deviation in the last test. Depth seems to performs very well except where there is initial noise to contend with. Variance was the slowest to converge in all cases.

References

[1] Ron Begleiter, Ran El-Yaniv, and Golan Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, December 2004.

[2] Tsutomu Kawabata and You Yanagisawa. Redundancy of symbol decomposition algorithms for memoryless source. *International Symposium on Information Theory*, pages 500–504, September 2005.