UNIVERSITY OF CALIFORNIA, SAN DIEGO

Normal Hedge in weighted trees for binary sequence prediction

A Thesis submitted in partial satisfaction of the
requirements for the degree
Master of Science

in

Computer Science

by

Andrea Biaggi

Committee in charge:

Professor Yoav Freund, Chair
Professor Sanjoy Dasgupta
Professor Charles Elkan

2010

The Thesis of Andrea Biaggi is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____
Chair

University of California, San Diego

2010

DEDICATION

To my family.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

Thanks to professor Yoav Freund and Sunsern Cheamanunkul for their help in assisting me with the thesis.

ABSTRACT OF THE THESIS

Normal Hedge in weighted trees for binary sequence prediction

by

Andrea Biaggi

Master of Science in Computer Science

University of California, San Diego, 2010

Professor Yoav Freund, Chair

This thesis presents the use of an online learning Hedging technique to predict patterns in a binary sequence. It is compared to previous techniques and this approach has faster learning rates and for some sequence it has better results in the prediction.

Chapter 1

Mind Reading Machines

1.1 History

1.1.1 Poe

Edgar Allan Poe inspired the work on mind reading machines in his short story *The Purloined Letter* [Poe80]. In this short story, an eight-year old boy played the "Odds and Evens" game with his friends and made a small amount of money. The game players were a guesser, the boy, playing against an opponent, in this case one of his friends. The game played by them was very simple: the guesser predicts the opponent's next move, the opponent chooses the next move as *even* or *odd*, each play had a bet of a small amount of money. If the guesser guessed correctly, he will win the money, otherwise the opponent will win the money. In the story, the guesser can determine what the opponent will do next.

1.1.2 Shannon

Claude E. Shannon [Sha53] proposed a machine approach to Poe's guesser. This solution is a simplified version of D.W. Hagelbarger's machine presented in Section 1.1.3.

In this machine, the player has to choose between q *right* or *left*. The machine was built with electric relays. Figure 1.1 displays how the circuit was implemented.

The machine looks for patterns in human behavior and assumes these patterns will be followed. Machine goes random (using a rotating commutator) before finding a pattern repeated at least twice by the player. The machine remembers what happened in the past in this specific pattern, noting if the player played the same and if it is a repetition of the behavior of the player.

It is possible to beat the machine with a ratio 3:1 by keeping track of all the cells in the machine. The strategy for winning is repeating a behavior pattern twice, and then changing the pattern.

1.1.3 Hagelbarger

D. W. Hagelbarger's [Hag56] machine had different inputs, instead of *right* or *left* it, had + or -. The machine recognized periodic sequences. The machine was more complex, but it was still built with electric relays. A diagram on how the machine is built can be found in Figure 1.2.

The machine follows two assumptions: the play of the person will not be random, and the machine is hard to beat. Computations are made whether the play should be *same* or *different* from the last one. The *state of play* of the machine that is determined by 3 factors:

1. if last play was a win or a lose
2. if second last play was a win or a lose
3. if the player played the same or different

The memory stores information at the beginning of every play:

1. Should the machine play the same or different in this state for winning? Controlled by a reversible counter $[-3, +3]$. If the machine should have played the same, the counter incremented; if not, decremented. (It forgets ancient history)
2. Has the machine been winning in this state? Remembering if machine won both, one or neither of the last 2 plays.

The machine play is determined by the following rules:

1. If the machine has lost the last 2 times in the state, then play random.
2. If the machine won one in the state, then with probability 3/4 follow the counter.
3. If the machine won both, then follow the counter.

Beating the machine is possible. The player knows the *state of play*, the odds in memory register and the sign of the counter. The player can force the machine to go to a certain state. A *sub-state* is determined by count in the counter and the odds. The machine's behavior in any sub-state do not depend on what the player did previously in it. The player can do always the same thing and does not depend on the random element.

1.2 Previous work

1.2.1 Mind Reader Applet

The sequence prediction we want to improve is a Java Applet done by Anup Doshi (Figure 1.3); this applet implements Weighted Context Tree [HS95] (an extension to Variable-Length Markov Models) combined with Dutch trees. This applet collected a lot of binary sequences we can use in order to understand where the approach of Weighted Context Tree fails and try to improve it with another algorithm.

1.2.2 Weighted Context Tree Algorithm

Helmbold and Schapire [HS95] proposed an algorithm to predict nearly as well as the best pruning of a decision tree. The work is done on Cesa-Bianchi et al.'s [CBFH$^+$97] idea on how to use expert advice.

Cesa-Bianchi's approach is to have as many prediction trees as the number of possible pruning of a starting prediction tree. At the beginning, each tree has a uniform weight of prediction. Weights of the trees are updated every time a feedback is received by giving more weight to the pruning tree that had a better prediction. This algorithm is computationally infeasible because the number of pruning trees is enormous.

Helmbold and Schapire proposed an efficient solution to the computationally intractable of Cesa-Bianchi et al. Instead of having a huge number of weighted pruning trees, there is a unique tree, that grows by exploration, with weights in the nodes approximating all the possible pruning trees. This approach uses less memory and also has faster updates, they are done on the interested branch and not in all the trees.

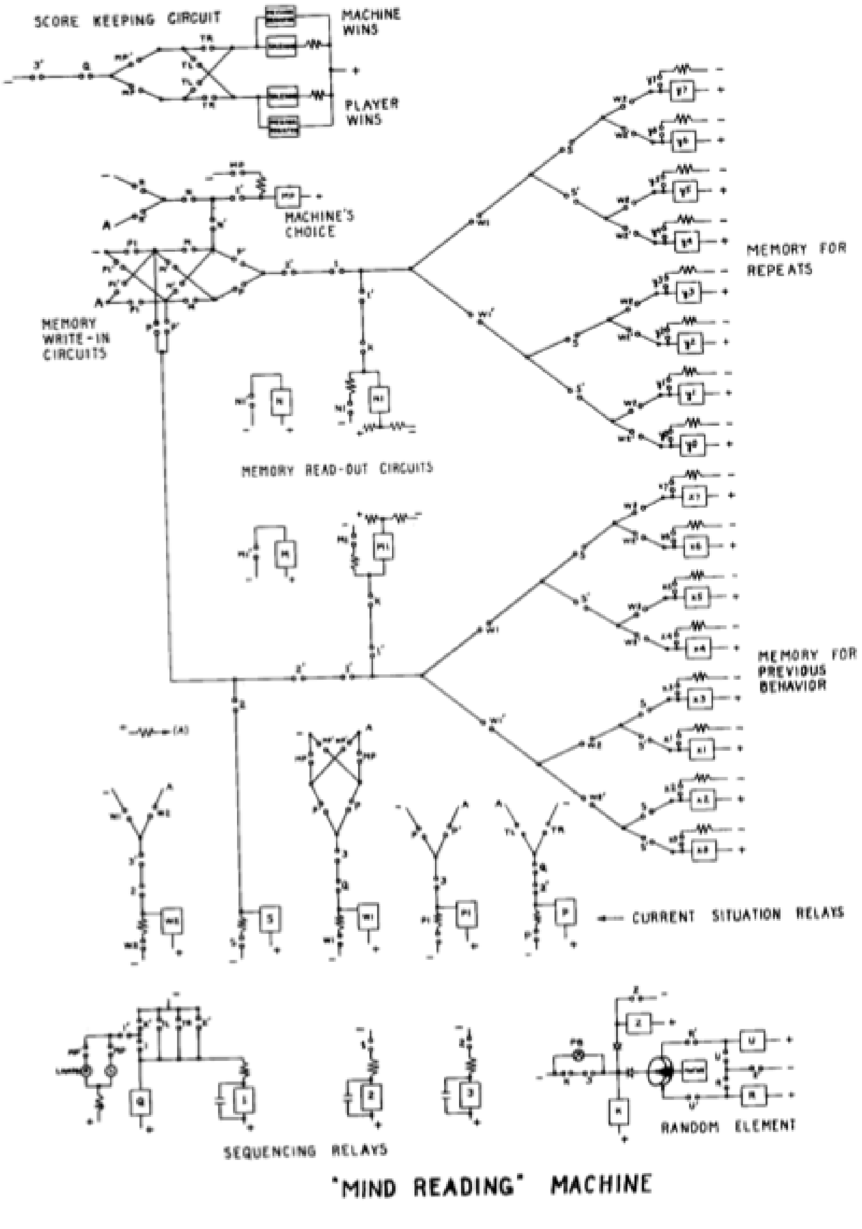


Figure 1.1: The relays of the Mind Reading machine proposed by Shannon

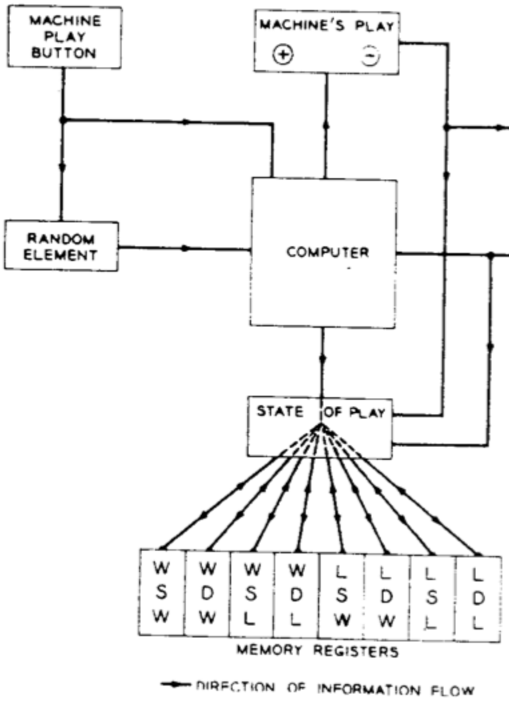
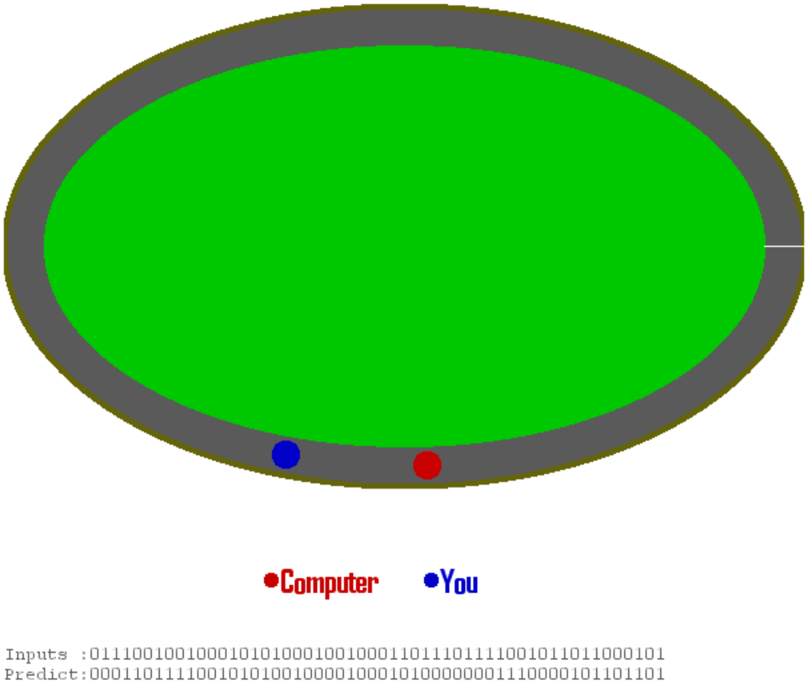


Figure 1.2: Block diagram of SEER by Hagelbarger



```
Inputs :011100100100010101000100100011011101111001011011000101
Predict:000110111110010101001000010001010000000011100001011011 01
```

Figure 1.3: Screenshot of the Mindreader Java Applet done by Anup Doshi

Chapter 2

Algorithm

2.1 Hedging Algorithms in Online Learning

Hedge is a master algorithm that has other algorithms called experts or heuristics. Those experts make predictions based on some input. Hedge assigns weights to each expert, the weight represents the level of trust the master has on that particular heuristic. The better the expert performs, the higher the weight o. The ultimate goal of hedge is to perform almost as well as the best expert in hind-sight. In the Online learning framework, at each time step, the master algorithm makes a prediction based on the weights of the heuristics; based on that prediction, the master algorithm and the experts suffer a loss. The regret of the master algorithm over an expert is the difference between them (loss of the master minus loss of the expert). The goal of hedge in assigning weight is to minimize the regret with the best expert.

2.1.1 Hedge(η)

Hedge(η) is a parametric hedging algorithm proposed by Yoav Freund and Robert E. Schapire[FS99, FS95]. Algorithm 1 summarizes Hedge(η)

Algorithm 1 Hedge(η)

```
$L_{i,0} \leftarrow 0, \forall i$ {cumulative loss}
initialize $w_{i,1}, \forall i$ {weight distribution, $\sum_{i=1}^{N} w_{i,1} = 1$}
for $i = 1, 2, \ldots$ do
    Each action $i$ incurs a loss $l_{i,t}$
    $L_{i,t} \leftarrow L_{i,t-1} + l_{i,t}$ {expert loss}
    $w_{i,t} = w_{i,1} \cdot e^{-\eta L_{i,t}}$
    $p_{i,t+1} \leftarrow \frac{w_{i,t}}{\sum_{j=1}^{N} w_{j,t}}$ {update probability distribution}
end for
```

2.2 Normal Hedge

We use Normal Hedge [CFH09], a parameter-free Hedging algorithm. It can be seen as Hedge(η) where the parameter η tunes itself while the algorithm is learning. Algorithm 2 shows the pseudocode of Normal Hedge.

Algorithm 2 NormalHedge

```
$R_{i,0} \leftarrow 0, \forall i$ {cumulative regret}
$w_{i,1} \leftarrow 1/N, \forall i$ {weight distribution}
for $i = 1, 2, \ldots$ do
    Each action $i$ incurs a loss $l_{i,t}$
    $l_{A,t} \leftarrow \sum_{i=1}^{N} w_{i,t} l_{i,t}$ {master loss}
    $R_{i,t} \leftarrow R_{i,t-1} + (L_{A,t} - l_{i,t}), \forall i$ {cumulative regret}
    Find $c_t$ satisfying: $\frac{1}{N} \sum_{i=1}^{N} \exp\left(\frac{([R_{i,t}]_+)^2}{2c_t}\right) = e$ {Algorithm 3}
    $w_{i,t+1} \propto \frac{[R_{i,t}]_+}{c_t} \exp\left(\frac{([R_{i,t}]_+)^2}{2c_t}\right)$ {update distribution}
end for
where $[x]_+ = \max\{0, x\}$
```

2.2.1 Binary search for finding c_t

In order to find the parameter c_t needed in the Normal Hedge algorithm we used binary search. We have two parameters, $c_{\min}$ and $c_{\max}$. We set those two values to be the endpoints of the range where we expect c_t to be. A priori, we know that $c_t > 0$ and $c_t < \infty$, therefore we could set $c_{\min} = 0$ and $c_{\max} = x$, where x is the largest representable number by our programming environment. The search of c_t is presented in Algorithm 3. If we bound c_t to plausible values during the algorithm initialization, the search converges very quickly.

Algorithm 3 find c_t

```
$c_{\min} \leftarrow 0$
$c_{\max} = x$ {where $x$ is the largest representable number }
while $c_{\min} \not\approx c_{\max}$ do
    $c_{test} \leftarrow \frac{c_{\min}+c_{\max}}{2}$
    $LHS \leftarrow \frac{1}{N}\sum_{i=1}^{N} \exp\left(\frac{([R_{i,t}]_+)^2}{2c_{test}}\right)$
    if $LHS < e$ then
        $c_{\max} \leftarrow c_{test}$
    else
        $c_{\min} \leftarrow c_{test}$
    end if
end while
$c_t \leftarrow c_{test}$
```

2.3 Normal Hedge in Weighted Context Tree

2.3.1 Data structure

Our input data structure is a binary sequence. At each iteration the algorithm receive an outcome; this outcome is appended to the end of the sequence that is needed by our algorithm in order to update itself and make predictions. An example of the sequence can be seen in Table 2.1.

Table 2.1: Example sequence received by the algorithm, prediction of x_t has to be made

...	x_{t-4}	x_{t-3}	x_{t-2}	x_{t-1}	x_t
...	0	1	1	0	?

For our approach we use a binary tree. Each node can have 0 or 2 children.

In each node there is an instance of Normal hedge with experts. The tree represents the previous history in the sequence, for example the root node (L0) represents no history (it can be seen as unigrams in language model); level 1 (L1) represents bigrams of history, and so on. All the nodes have tree experts, except the leaves have only two. The experts are:

1. Always predict 0. ($p_0 = 0$)
2. Always predict 1. ($p_1 = 1$)
3. Use prediction of the subtree following the path of x_{t-l-1} where l represents the level the node is in, in the case of a non leaf.

Each expert has a weight associated with it used for the prediction; it is initialized to $1/n$ where n is the number of experts in that specific node. A created tree with depth 2 can be seen in Figure 2.1.

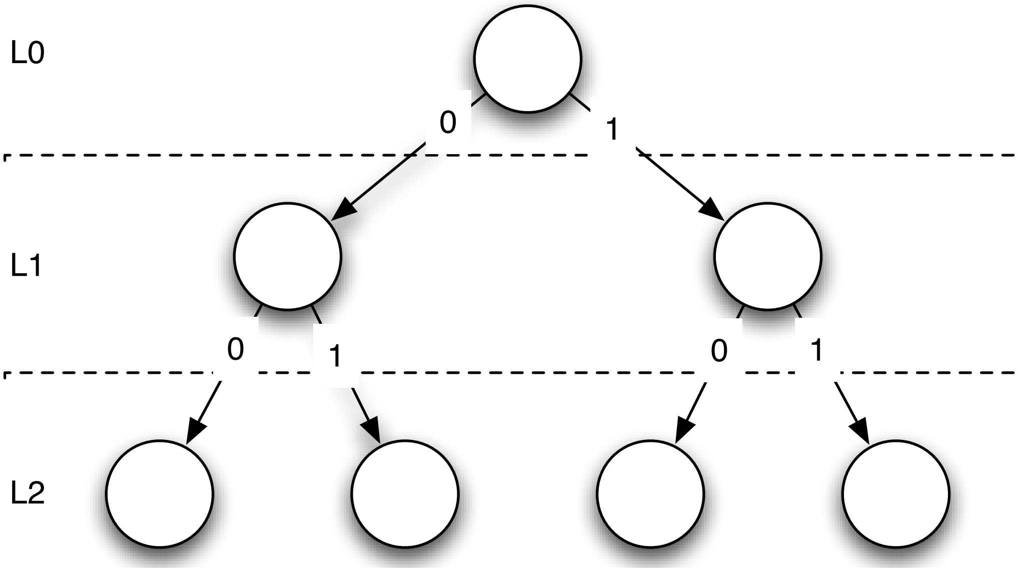


Figure 2.1: A Normal Hedge weighted context tree of depth 2

2.3.2 Prediction

In order to predict we have to use the information along the path of the sequence. For the example input presented in Table 2.1, Figure 2.2 shows the path of the nodes considered in the prediction. The algorithm for prediction is presented in Algorithm 4 in which $n = t - 1$. In order to give a prediction that is either 0 or 1, we execute the algorithm as $p \leftarrow$predict(root,$x_1, x_2, \ldots, x_n$). The guess is chosen by a random coin flip to be 1 with probability p and 0 with probability $1 - p$.

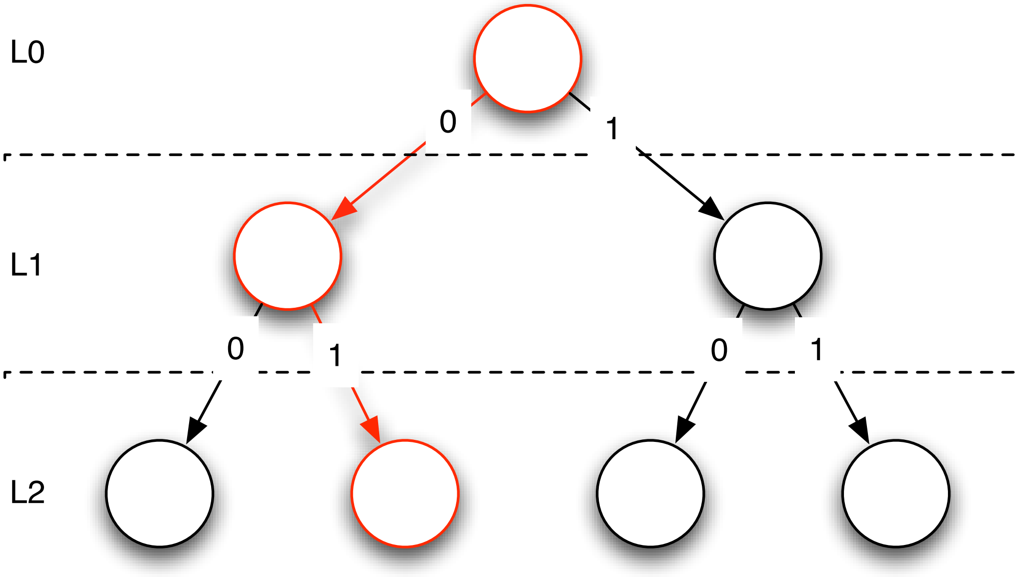


Figure 2.2: A weighted context tree of depth 2 with history path highlighted

2.3.3 Update

Once the input of the new element of the sequence is received from the user we have to update our weights. We refer the new element of the sequence as outcome. We update the tree with Algorithm 5 applied to the root node. If node is a leaf, in the second line of the algorithm, we do not consider $w_2 \cdot l_2$.

Algorithm 4 predict(node,$x_1, x_2, \ldots, x_n$)

```
if $node.firstVisit$ then
  return $1/2$
else if $node.isLeaf$ then
  return $w_0 \cdot 0 + w_1 \cdot 1$
else
  if $x_n = 0$ then
    next node ← child at 0
  else
    next node ← child at 1
  end if
  return $w_0 \cdot 0 + w_1 \cdot 1 + w_2 \cdot$ predict(next node,$x_1, x_2, \ldots, x_{n-1}$)
end if
```

Algorithm 5 update(node, outcome, $x_1, x_2, \ldots, x_n$)

```
$l_i \leftarrow |\text{outcome} - p_i|, \forall i$
$l_A \leftarrow w_0 \cdot l_0 + w_1 \cdot l_1 + w_2 \cdot l_2$
$R_i \leftarrow R_i + (L_a - l_i), \forall i$
find $c_t$ according to Algorithm 3
$p_i \leftarrow \frac{\frac{[R_i]_+}{c_t} \exp\left(\frac{([R_i]_+)^2}{2c_t}\right)}{\sum_{j=0}^{2} \frac{[R_j]_+}{c_t} \exp\left(\frac{([R_j]_+)^2}{2c_t}\right)}$
if $node.isLeaf = false$ then
  if $x_n = 0$ then
    next node ← child at 0
  else
    next node ← child at 1
  end if
  update(next node, outcome, $x_1, x_2, \ldots, x_{n-1}$)
end if
```

Chapter 3

Experiments

3.1 Experiments only with root node

In this set of experiments we use only a single node of the tree, the root node. Since it is a leaf, it has only two experts: always predict 0 and always predict 1. In the experiments three algorithms are compared: NormalHedge, Hedge(η) with different parameters and a greedy algorithm. The greedy algorithm predicts according to the majority, the probability to predict 1 is 1 if the counts of 1's in the sequences are greater than the counts of 0's, it predicts 0 if the count of 0's is greater than the count of 1's; in case of same count an unbiased coin is flipped to decide the prediction. Accordingly the weight of predicting 1 is decided by the relation between counts of 1's and 0's, if the count is equal the weight of predicting 1 is 1/2. These experiments are done to see how the algorithms respond to a synthetically generated data. The two metrics used to compare the three algorithms are the weight given to the expert suggesting to predict one and how much regret the algorithm suffers from the prediction of the best expert. We discarded to show graphs about the loss because the loss can be inferred from the regret.

3.1.1 Sequence length variation

The first experiment is designed to analyze how the algorithm evolves during each input received by it. In this experiment we have 6 algorithms: NormalHedge, Hedge(1/8), Hedge(1/4), Hedge(1/2), Hedge(1) and greedy. We average the result over 100000 runs. Before each run a sequence is generated with p_1 probability to generate 1 at each position in the sequence, then this sequence is used as same input for all the 6 algorithms. All the data needed for the two metrics is collected and averaged over the number of runs.

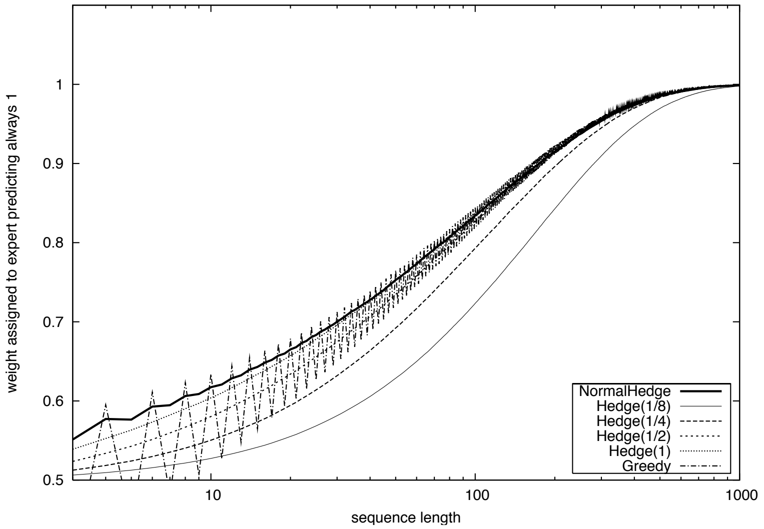


Figure 3.1: How does the weight of always predicting 1 change over sequence length. With fixed probability $p_1 = 0.55$ to generate 1, averaged over 100000 runs. Normal Hedge is more aggressive in deciding to predict 1, compared to the other strategies.

As we can see from the graphs in Figure 3.1 and Figure 3.2 NormalHedge learn faster the correct weight, that is the most frequent outcome in the sequence, in respect to Hedge(η) but eventually all the six algorithms will converge to the correct value. The greedy algorithm learns as fast as Normal Hedge the correct weight of predicting always 1, especially when the sequence is generated with a probability close to 0 or close to 1. When the probability is not clear the greedy algorithm makes a lot of mistakes in the prediction.

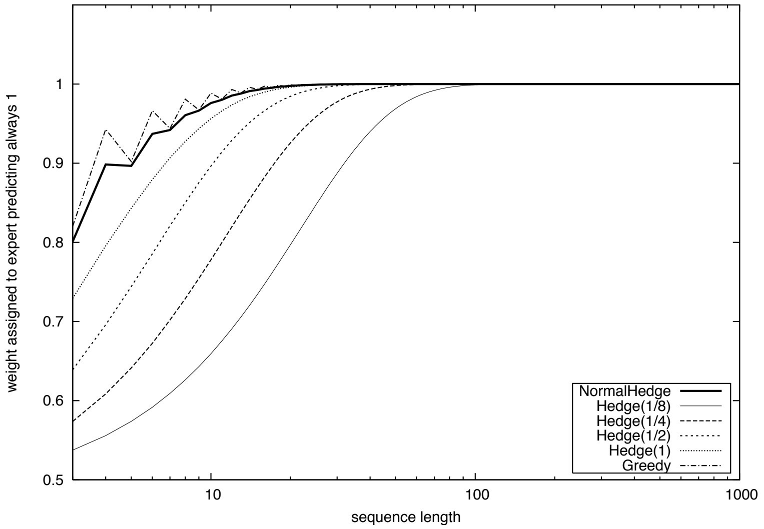


Figure 3.2: How does the weight of always predicting 1 change over sequence length. With fixed probability $p_1 = 0.8$ to generate 1, averaged over 100000 runs. Normal Hedge and greedy are more prone to put the weight to predicting 1 early in the sequence.

Another way to better see the differences between the algorithms is to compare the cumulative regret the algorithm suffers during the input. The regret is computed by the difference between the prediction of the algorithm, based on the weighted average of the experts, and the best expert.

From Figure 3.3 and Figure 3.4 we can see that the Normal Hedge algorithm suffers overall less regret compared to the Hedge(η). In the case where $p_1 = 0.55$, Normal Hedge and Hedge(1) have almost the same cumulative regret. All the other experts suffer more regret because they take longer to understand with which probability the sequence is generated due to their lower learning rate. When the probability is more near to the extremes Normal Hedge (for example $p_1 = 0.8$) suffers of less regret in respect to all the other algorithms. The results are explained because Normal Hedge discovers more quickly the best expert and all the weight is assigned to it. The slowness of deciding the best expert is directly translated to more regret from the algorithm.

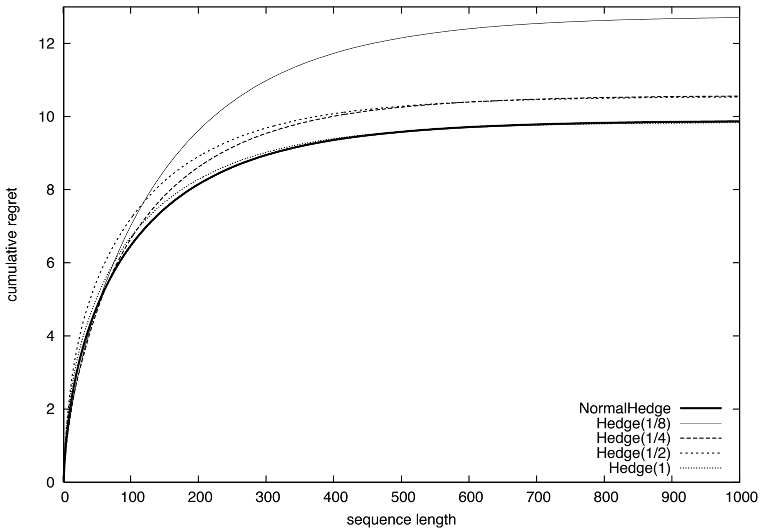


Figure 3.3: Cumulative regret of the algorithm over sequence length. With fixed probability $p_1 = 0.55$ to generate 1, averaged over 100000 runs. Normal Hedge and Hedge(1) suffer a similar algorithm regret. All the other algorithms have bigger cumulative regret with respect to the former two.

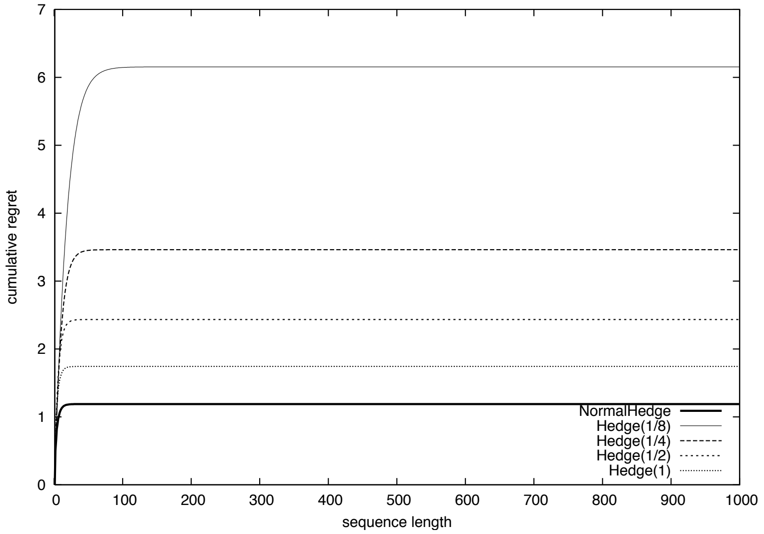


Figure 3.4: Cumulative regret of the algorithm over sequence length. With fixed probability $p_1 = 0.8$ to generate 1, averaged over 100000 runs. Normal Hedge performs clearly better than the other algorithms.

3.1.2 Generator variation

The second experiment in this set is done similar to the first one, but instead of recording the weight of the algorithm to predict 1 and the cumulative regret over the sequence length, we keep track of them over the different probabilities p_1 of the generator of the sequence and the result is acquired at the outcome of the last element in the sequence generated. Like before, we have 6 algorithms and the result is averaged over 100000 runs. In this case we use a default sequence length of 200. This length is chosen because it is the maximum length of a Mind Reader instance.

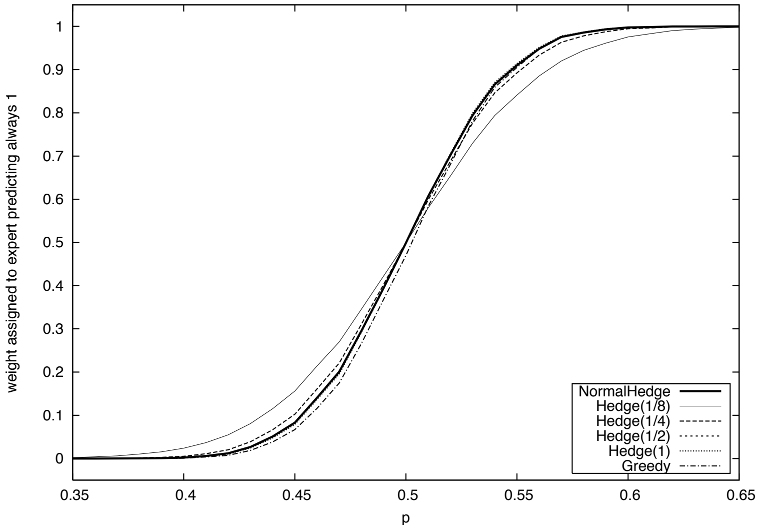


Figure 3.5: How does the weight of always predicting 1 change based on probability p_1 to generate 1. With fixed sequence length, averaged over 100000 runs. All the algorithms behave similarly.

Figure 3.5 shows that Normal Hedge, Hedge(η) and greedy behave in a very similar manner. But Normal Hedge has an advantage over Hedge(η) with η smaller than 1. In the case of the current mind reader setting means that with the input generated Normal Hedge will perform generally better then Hedge(η). Greedy algorithm perform slightly better than Normal Hedge.

In order to understand better how the algorithms react by changing the probability of the generator of the sequence we use the regret metric similar to the prior set of experiments.

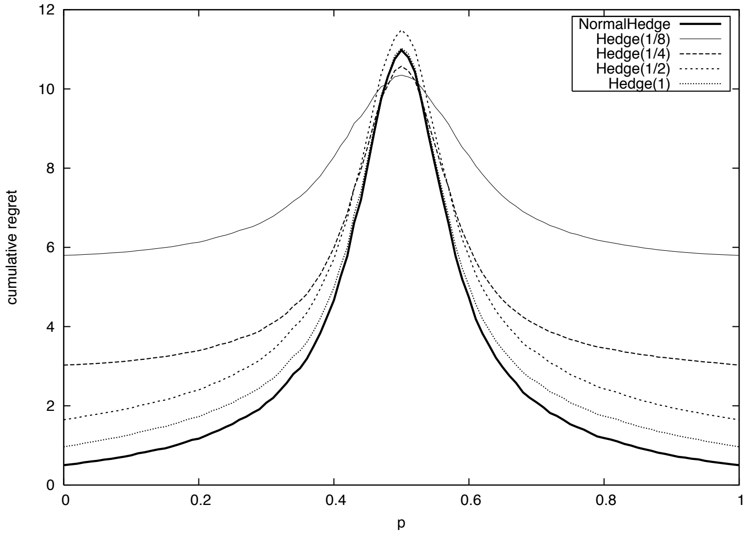
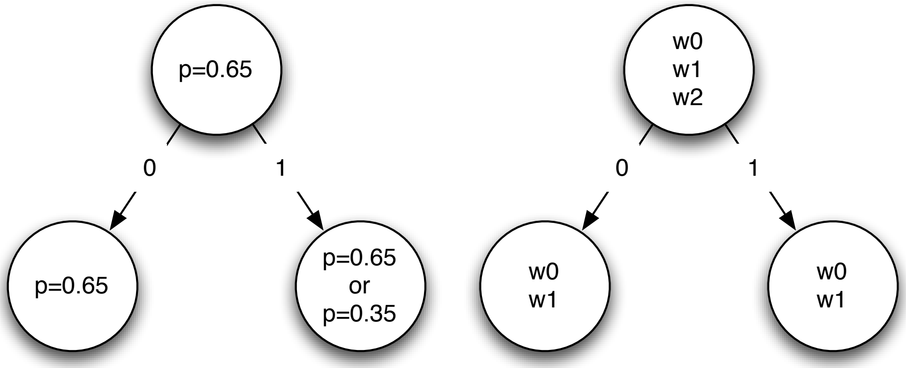


Figure 3.6: Cumulative regret of the algorithm over probability p_1 to generate 1. With fixed sequence length, averaged over 100000 runs. Overall Normal Hedge suffers less regret than all the other algorithms.

The results are summarized in the graph in Figure 3.6. From the results we can conclude that Normal Hedge does overall better in respect to the other algorithms. Some instances of Hedge(η) have less regret in the area around $p_1 = 0.5$, but outside of the small portion of best performance they are considerably worst than Normal Hedge. Hedge(η) with a low value of η have less regret when the probability of the generator is close to 0.5. This is because the algorithm does not commit quickly on what the best expert will be, and thus making less mistakes.

3.2 Experiments using trees

In this set of experiments data is generated by a Variable Length Markov Model Tree. We use the tree in Figure 3.7(a) to generate the elements in the sequence. Each leaf of the Variable Length Markov Model has a p probability to generate 1 and a $1-p$ probability to generate 0. As in the Normal Hedge tree nodes represent history. For example $w1$ in the child following the path 0, represents the probability of generating 1 when the previous bit is 0. A tree of the shape in Figure 3.7(b) is used to predict the outcome.



(a) Generation tree

(b) Prediction tree

Figure 3.7: Tree used to generate synthetic data with the two configurations, and Normal Hedge Tree with weights used for predicting.

The experiment in the Variable Length Markov Model tree structure is to see how the weights of an expert in an Hedging algorithm change during time. In the particular scenario given in Figure 3.7, the most interesting node to analyze the changes is the root node because all the three weights are affected. Therefore, our experiment is to study the change of the weights in the root for Normal Hedge. We use for simplicity a tree with depth 1. We use two configurations of the generating tree. One is when all the children have the probability $p = 0.65$ of generating 1, this is called the *same configuration*. The other is having one child leading to predicting 0 and the other leading to predicting 1, this is called the *different configuration*; in our particular case we use the children following the path 0 to predict with $p = 0.65$ and the children at 1 to predict with $p = 0.35$.

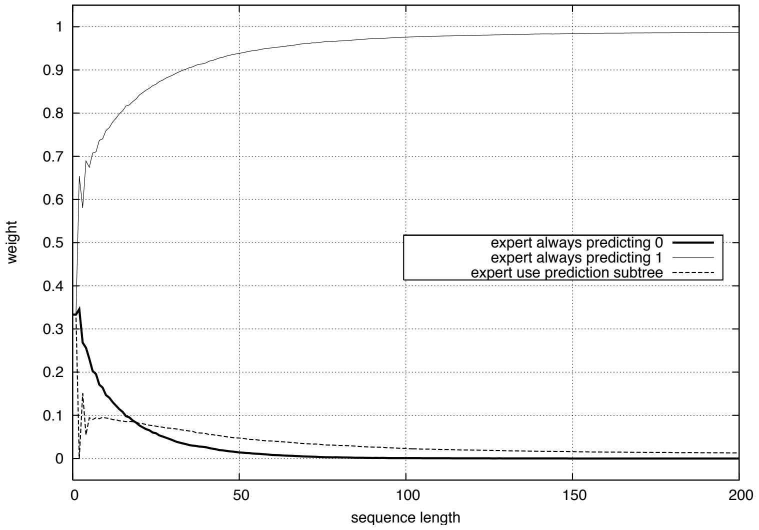


Figure 3.8: Experts weights change with data generated with *same configuration*, averaged over 10000 runs. All the weight is given to the expert predicting 1 in the root node because it is the most relevant choice.

From the results in Figure 3.8 and Figure 3.9 we can see that the Normal Hedge Tree understands the behavior of the generating tree and the weights adapt to better predict the sequence. In the case of the *same configuration* (Figure 3.8), in the root node, the weight assigned to go on the next level (expert 2) is close to 0 because it is not necessary to exploit the history in predicting the outcome. On the other hand, in the scenario of *different configuration* (Figure 3.9), Normal Hedge Tree learns the importance of using one bit of history to understand the generator configuration, but this information is learned slower with respect to the same configuration.

Since the update and prediction of Normal Hedge Tree is recursive, this behavior of predicting the generating tree can be seen also for bigger generating trees with branches of different lengths.

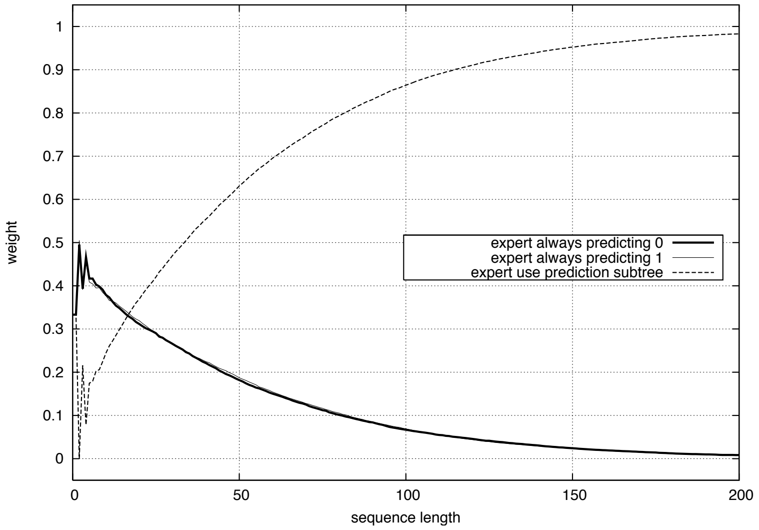


Figure 3.9: Experts weights change with data generated with *different configuration*, averaged over 10000 runs. The expert suggesting to use the subtree to predict the outcome receives most of the weight understanding the shape of the VLMM generator tree.

3.3 Experiments with Mind Reader data

Anup Doshi's Mind Reader collected relevant data in logs throughout the Mind Reader life span. The data is presented in the following form:

```
/<IP>:<ENTRY_ID>: <USER>: <TIMESTAMP>
<ALGORITHM>
<USER_SCORE>
<HIGHEST_WEIGHT>
P:<PREDICTED_SEQUENCE>
I:<USER_SEQUENCE>
```

The score is given by starting from 0. If the algorithm guesses the user input, -1 is added to the score, otherwise +1 is added to the score. An entry example (with a changed IP for privacy reasons) can be seen as:

```
/241.4.12.135:3586: Raphael: Mon Jul 13 15:19:08 GMT-08:00 2009
VarTreeExpert
-31
26688.0
P:00011010111100101011000101110010110111010100110110001000001001...
I:01101100010001110101010100111000010001100011000110001111110001...
```

In our experiment, we run Normal Hedge Tree over the sequences collected from the logs. We run the algorithm for different depths of the tree and see how it compares with the score. The score of Mind Reader depends on some randomness, therefore we run the algorithm 100 times and we compute the average score in order to have more reliable data. The log data had a total of 11391 wins of the algorithm against the user. When we run the Mind Reader algorithm we found 11170 sequence with an average user score below 0; surprisingly, the average difference between the log score and the computed score was that the log score was lower by 5.44 with respect to the computed score. Normal Hedge is run multiple times and we keep track of the average score over 100 runs. We tested some high score sequences in the same applet and we got scores that were lower than the original score.

Since all the entries are present in the logs, also when only 2 input bits are given, we discard all the entries that are less than 100 bit long. The entries we use in the test are therefore 13189 over the 18252 entries in the log.

The concept of winning for the algorithm is when the user has a final score of smaller than 0. One algorithm performs better than another if the user score in the algorithm is lower than the other. Table 3.1 collects the results of the experiment.

Table 3.1: Comparison between Normal Hedge Tree (NHT) and current Mind Reader (MR) performance over 13189 sequences

depth	MR wins	NHT wins	NHT better score than MR	NHT wins and MR loses	NHT loses and MR wins
0	11170	6834	2062	233	4569
1	11170	8953	3484	202	2416
2	11170	9566	4676	192	1796
3	11170	9693	4999	191	1668
4	11170	9708	5063	189	1653
5	11170	9706	5057	187	1652
6	11170	9716	5060	190	1646
7	11170	9709	5076	195	1657
8	11170	9721	5082	193	1644
9	11170	9729	5070	192	1630
10	11170	9713	5069	192	1645

From the results we can see that Normal Hedge Tree wins less times the user with respect to the Mind Reader algorithm. But for some sequences, Normal Hedge Tree beats the user but Mind Reader does not. We can see that for those particular entries Normal Hedge performs better with a depth of 4 or 5. This results suggest to combine the two algorithms with Normal Hedge. So that the new algorithm performs close as the best in hindsight.

3.3.1 Discounted regret

We propose to use discounted regret in the update rule in Algorithm 2. Instead of $R_i \leftarrow R_i + (L_a - l_i), \forall i$, the new update rule will look as follow:
$R_i \leftarrow \alpha \cdot R_i + (L_a - l_i), \forall i$. For this experiment we use a depth of the Normal Hedge Tree to be 4. We try different values of α averaged over 100 runs, and we collect data on how many Mind Reader entries the Normal Hedge Tree algorithm will win against the user.

Table 3.2: How many times Normal Hedge Tree with discounted regret α beats the user over 13189 entries

α	NHT wins
1	9708
0.9	10677
0.85	10718
0.8	10834
0.75	10873
0.7	**10973**
0.65	10906
0.6	10854
0.5	10727
0.4	10557
0.3	10401
0.2	10150
0.1	9513
0.0	6544

As we can see from Table 3.2 discounting the regret helps the Normal Hedge Tree in his prediction. This is probably because it helps to "switch" faster from the best expert at some time to the other best expert in a following time. But too much discount can mislead the algorithm and sticking with the best prediction in a particular time, with a small relevant history, instead of remembering the history for helping the prediction. The performance of Normal Hedge Tree with $\alpha = 0.7$ has a performance worst than Mind Reader by 1.49%.

3.4 Combining Experts

In this experiment we combine the Mind Reader algorithm with Normal Hedge Tree, both as experts in a Normal Hedge algorithm. We try some relevant discount factors to see how the combination of experts performs. The experiment is averaged over 100 runs.

Table 3.3: Comparison between the discounted combination (CE) of Normal Hedge Tree and Mind Reader versus current Mind Reader (MR) performance over 13189 sequences

α	MR wins	CE wins	CE better score than MR	CE wins and MR loses	CE loses and MR wins
1	11170	10582	2105	499	958
0.8	11170	10778	4892	567	958
0.75	11170	10809	5046	621	980
0.7	11170	10803	5104	620	986
0.65	11170	10795	5103	643	1018

We expected to have a better performance compared to Mind Reader, but the results in Table 3.3 show otherwise. This problem may be because the combined Normal Hedge takes some time before deciding which expert to follow. At the time of decision the combined Normal Hedge predicts using both experts and it can lose some points against the user.

Chapter 4

Conclusions

From the experiments we can derive some conclusions. Using Normal Hedge in Weighted Context Trees is more effective and aggressive than Hedge(η) for synthetically generated data, but it recognizes faster and better patterns and not human input. Normal Hedge performs worse than Mind Reader on the log data. In some sequences Normal Hedge can predict the user input and succeed, where Mind reader fails. This suggests that a good combination between the two algorithms, using Normal Hedge and having as experts the two analyzed approaches will perform close as the best of the two in hindsight. Using a combination of the experts we discovered that the results are not exactly as expected. The combination does not perform better to either of the experts taken singularly.

Chapter 5

Future Work

In the near future we will substitute the current Mind Reader algorithm with a choice for the user to compete against 3 algorithms: the already implemented Mind Reader, Normal Hedge Tree with discounted regret, and the combination of the two with Normal Hedge.

A second application could be to integrate our algorithm in Automatic Cameraman, an interactive display in the Computer Science and Engineering building at UCSD.

Another application, that we will create to collect more data, is a mobile phone application, most likely an Android app, since Android apps are written in Java, to extend the product to a broader group of people around the globe.

Appendix A

Implementation

A.1 Programming Language

This thesis is implemented in Java(TM) SE Runtime Environment. The choice of Java is given because Mind Reader is written in Java and the goal of this thesis is to provide an algorithm to improve it. Therefore, if our algorithm is written in Java it can easily substitute the current Mind Reader algorithm, or better it can be combined to exploit the strength of both.

A.2 Class Organization

The class organization of Normal Hedge with trees is relatively simple. We have a Node abstract class where Normal Hedge is implemented. Two subclasses extend the Node class because their behavior of update and predict differs whether a node is a leaf or not. Another class called Tree contains the Nodes, this class can be seen as the Master algorithm, even though all the computations start and occur in the root node. An interface *Expert* defines all the actions an expert should implement. This interface can be used for extensibility by adding more experts to the nodes or to the tree in order to have more heuristics for the prediction. The class diagram of the thesis can be seen in Figure A.1.

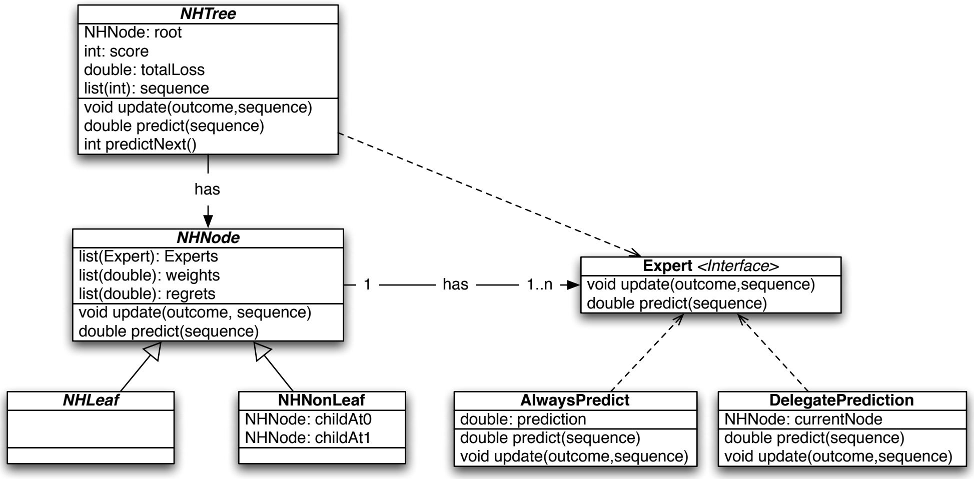


Figure A.1: Class diagram of Normal Hedge in Weighted Context Tree

A.3 Programming Tricks

Some programming tricks for avoiding numerical instability were suggested by Sunsern Cheamanunkul. This numerical instability is given by the numerical approximation of the binary representation of numbers smaller than one. We defined a constant for the precision we want as: $\epsilon = 10^{-9}$. We use ϵ in two parts of Normal Hedge to ensure numerical stability:

1. while looking for c_t, in Algorithm 3, we want to find when $c_{\min} \approx c_{\max}$, this approximation in our program is written as $|c_{\max} - c_{\min}| < \epsilon$
2. in Algorithm 2 if before normalizing the weights $\sum_{i=0}^{N} \frac{[R_i]_+}{c_t} \exp\left(\frac{([R_i]_+)^2}{2c_t}\right) < \epsilon$, we set all the weights to be $w_i = 1/N$

Another trick used is to give an upper bound to the binary search. This makes the computation faster and the c_t we want to find is always upper-bounded by this value. The upper-bound is computed: $c_{\max} = \sum_{i=0}^{N} [R_i]_+^2$

Bibliography

[CBFH$^+$97] Nicolò Cesa-Bianchi, Yoav Freund, David Haussler, David P. Helmbold, Robert E. Schapire, and Manfred K. Warmuth. How to use expert advice. *J. ACM*, 44(3):427–485, 1997.

[CFH09] Kamalika Chaudhuri, Yoav Freund, and Daniel Hsu. A parameter-free hedging algorithm. *CoRR*, abs/0903.2851, 2009.

[FS95] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting, 1995.

[FS99] Yoav Freund and Robert E. Schapire. Adaptive game playing using multiplicative weights. In *Games and Economic Behavior*, pages 79–103, 1999.

[Hag56] D Hagelbarger. Seer, a sequence extrapolating robot. *I.R.E. Trans. Electronic Computers*, 5:1–7, 1956.

[HS95] David P. Helmbold and Robert E. Schapire. Predicting nearly as well as the best pruning of a decision tree. In *Machine Learning*, pages 61–68, 1995.

[Poe80] Edgar Allan Poe. *The Purloined Letter (Tale Blazers).* Perfection Learning, 1980.

[Sha53] C Shannon. A mind-reading machine, bell laboratories memorandum. In *Reprinted in the Collected Papers of Claude*, pages 688–689. IEEE Press, 1953.