A.M. van Eden

Breaking NormalHedge

Bachelor Thesis

Thesis Supervisor: Prof. Dr. P. D. Grünwald

Date of Bachelorexamination: 14th August 2014

Mathematisch Instituut, Universiteit Leiden

Abstract

In the past few years there has been a great development in the field of sequential prediction. Starting with the simple, yet often effective, Follow-The-Leader strategy, numerous different strategies have been conceived. The most prevalent algorithm being used to realize these strategies is Hedge. This algorithm's performance crucially depends on a parameter called the learning rate.
Based on the work of Cesa-Bianchi, Mansour and Stoltz, a better Hedge algorithm named AdaHedge has been developed by Grünwald, De Rooij, Van Erven and Koolen that has great worst-case performance bounds. It sets the learning rate parameter dynamically without using the doubling trick. This means that it looks at the previous results to make the next prediction.
At around the same time, a new, completely different type of algorithm, named NormalHedge, has been devised in San Diego. NormalHedge is parameter free. This algorithm, by Freund, Chaudhuri and Hsu, completely skips the learning rate complication.
In some simple examples it has been shown that NormalHedge has similar if not better performances than all traditional Hedge strategies. In this paper AdaHedge and some similar other Hedge Algorithms are compared to NormalHedge. First we will do this through examples given by Grünwald et al. that we will reproduce. Next an extensive and elaborate data sequence is created. This complicated experiment will give new insights in the strengths and weaknesses of the algorithms.

Contents

1 Introduction

In this paper we look at the decision-theoretic online learning (DTOL) framework. This is a variant of the problem of *prediction with expert advice* introduced be Vovk [2]. The goal here is to sequentially predict a sequence of data.
We follow up on the work published by Freund and Schapire [1] who first introduced the Hedge setting. This was improved by Grünwald et al.[3] with the development of the AdaHedge and FlipFlop Algorithms. These algorithms were seen as the most effective in providing good results for both favorable and unfavorable data sequences. However, recently, a new type of algorithm has been developed, NormalHedge, which does not make use of the learning rate parameter like the Hedge algorithms.

Until now, all the cases tested with NormalHedge gave better performances than AdaHedge. Our goal is to find and test a complex situation where NormalHedge is expected to perform poorly in comparison to AdaHedge. Preferably this case would have AdaHedge perform well but this is not a necessity. More simply put, we want to find a data sequence where AdaHedge works well and NormalHedge doesn't and thus "break" NormalHedge.

First, an overview of the Hedge setting is given and explained. Then the most common Hedge algorithms and the new NormalHedge algorithm are described. Next, we try to reproduce the simple examples shown by Grünwald et al. [**?**] to check make sure the matlab implementations are correct. Finally, we introduce a complicated situation where the algorithms will be strained and then discuss the results.

2 Framework

In the Hedge setting, the person using the algorithm, which we shall call "Learner", must each round $t = 1, 2, \ldots, T$ make a prediction of the outcome in the next round. The Learner does this by referencing a set of K forecasters we call "Experts". The reader should note that the forecasters being named experts says nothing about their actual accuracy.
At each time, t, the Learner has access to a set of predictions made by these experts. To make his own forecast, the Learner assigns a nonnegative weight $w_{t,k}$ to each expert prediction. In addition, the sum of the weights should equal to one. For the sake of convenience, the weights are put in vector form, $w_t = (w_{t,1}, w_{t,2}, \ldots, w_{t,K})$.
The reader may think of the weights as determining the importance allocated to an expert's opinion. It is possible, for example, to put all your weight on one expert or to spread it out evenly over all experts. We will come back to this later.

After a prediction is made, the "environment" reveals the true value. Each expert then incurs a loss. The losses are also put together in a vector, $l_t =$

$(l_{t,1}, l_{t,2}, \ldots, l_{t,K})$. These losses all have values between 0 and 1.
The Learner's loss is the dot product $h_t = w_t \cdot l_t$. This is intuitively correct as the prediction incurred has a certain weight attached to the forecast of each expert. Therefore the loss incurred by each expert is also part of the loss incurred by the Learner, the proportion depending on the weight given. Because we look at how well an algorithm does over a period it makes sense to look at the cumulative losses. We denote these with a capital letter: $L_{t,k} = l_{1,k} + \cdots + l_{t,k}$ is the cumulative loss of expert k and $H_t = h_1 + \cdots + h_t$ is the cumulative loss of the Learner. The *regret* is how we denote the performance of the Learner and the goal is to have a regret that is a small as possible:

$$R_t = H_t - L_t^* \quad \text{with } L_t^* = \min_k L_{t,k}$$

What the regret signifies is the difference between the loss of the Learner and the best expert so far. This is the expert that has the smallest cumulative loss $L_T = \sum_{t=1}^{T} l_{t,k}$.
It is possible that the regret function decreases. In other words, the function is not monotonically increasing. A decrease at time t symbolizes the Learner performing better than the best expert so far.

3 Algorithms

The most common solution to the DTOL problem are Hedge algorithms. These are algorithms that all are dependent on one parameter: the learning rate η. How this rate is set has been the recent focus of most mathematicians working in this field. The most basic strategy is described in Figure 1. Here the learning rate is a constant.

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

Figure 1: Pseudo-code for the Hedge algorithm [7]

3.1 Follow-The-Leader

A simple strategy with a constant η is Follow-The-Leader (FTL). In FTL, all the weight is given to the best expert so far (or spread evenly if there are multiple experts with the same smallest loss). To implement FTL in the Hedge algorithm, the learning rate must be set to infinity. In the code, that is equivalent to setting η to a very large number.

FTL is a intuitive strategy as one could expect that the best expert so far has good odds of remaining so the next round as well. In addition this strategy has been used and proven its worth in certain Financial applications. However, very quickly, it is realized that FTL being a good option is very often not the case. This especially when the data is antagonistic. When this is the case, the regret of FTL kan even become linear following the line $y = \frac{1}{2}t$. This is worst result the Learner could possibly have and is thus the worst-case regret bound for FTL. We will illustrate this more in section 4.

3.2 SafeHedge

In response to this new challenge, methods for finding the value for η that give good regret bounds for all data sequences, thus being "safe", are devised. One of these methods that is safe is fittingly called SafeHedge. Here a learning rate is devised to optimize for the worst-case regret bound found by Cesa-Bianchi and Lugosi (section 3.7) [5].

$$\forall\ \eta > 0 : \quad R = \frac{\ln K}{\eta} + \frac{\eta T}{8}$$

What Cesa-Bianchi and Lugosi found, is that when using the equation above to find η the lower bound of the worst-case regret is $O(\sqrt{T \ln K})$. This means that if the algorithm performs the worst it can it wil be following the line $\sqrt{T}$ in the long run. The algorithm works "well", or has good results, for a certain data sequence when the regret is substantially lower than the worst-case regret bound. This definition of an algorithm's performance will be used in the whole paper.

What we can see is that SafeHedge needs information before the algorithm actually starts. This includes the length of the loss sequence T and the number of experts K. This prior needed knowledge makes it a less flexible alternative as this is not always available. Without these values, the learning rate cannot be found and the algorithm cannot be executed.
The SafeHedge algorithm does work well for "hard" data sequences, sequences that are similar to the worst-case. For "easier" data however, this bound is too weak. This results in a high regret when a low one is easily attainable.

What we may conclude is that setting the learning rate to a constant does not always give a good performance. On the contrary, for many data sequences, Hedge with constant η perform poorly. To solve this problem, dynamic methods of setting η have been developed.

3.3 Dynamic Learning rate

There are numerous different strategies for setting the learning rate dynamically.

One such strategy has been developed by Cesa-Bianchi, Mansour and Stoltz [8] (CBMS from now on). They did this by adjusting the learning rate each round based on the previous observations. So now, not only the length of the data sequence up till now is important, but also what all the actual losses of the experts are. How this is different from SafeHedge is that prior information like the length of the sequence T is not needed to make the prediction. With this technique, the CBMS algorithm was developed which is the groundwork for the next algorithm we will be looking at.

The CBMS algorithm was a big step in improving the Hedge algorithm. It gave good regret bounds for worst-case data. However in the past few years a refinement of this algorithm has been developed by Grünwald et al [3]. This new algorithm, AdaHedge, improves on the bounds by a factor 2. While not being a enormous improvement, this is reason enough to work with AdaHedge instead of the CBMS algorithm. Another improvement is that it exploits easy sequences of losses more aggressively thus performing better in these cases. What this means is that the actual regret difference may be larger than the factor 2 regret bound difference. The actual lower worst-case regret bound of AdaHedge is also $O(\sqrt{T \ln K})$. This means that it will not perform worse than SafeHedge in the worst-case. It is possible that SafeHedge performs better than AdaHedge but the results will always be below the lower bound. Grünwald et al. [3] found that AdaHedge works well when the weights concentrate fast to one expert which often happens when the data is easy or not very adversarial. On the other hand, AdaHedge works poorly when the
How AdaHedge is implemented in code is described in section 6.

3.4 A combination: FlipFlop

The last algorithm that is going to be experimented with from the Hedge family is the FlipFlop algorithm also by Grünwald et al [3]. This algorithm uses both the FTL strategy and the AdaHedge strategy. Every round FlipFlop looks at the data and makes a decision if it should use the one strategy or the other. It does this by carefully alternating between both algorithms and then looking which one gives a preferable result. This technique keeps the worst-case regret bound of AdaHedge up to a constant factor and its regret is also still bounded by the regret of FTL multiplied by a constant. This mean that the lower bound regret for the worst-case is also $O(\sqrt{T \ln K})$. What this also means is that this method will not, for each and every data sequence, produce the very best results but it does guarantee good results for all data sequences. This makes it a more robust algorithm. How this algorithm is implemented is shown in section 6.

3.5 NormalHedge

As mentioned before, a new solution to the DTOL problem has recently been developed by Chaudhuri et al [4]. This methode uses the same framework as Hedge but is parameter free. Similarly though to AdaHedge it uses the previous data to tune itself. This new strategy was inspired by the difficulties the basic Hedge algorithm has when there is a great amount of experts. In Figure 2 the pseudo code of the NormalHedge algorithm is given. One can see that the code starts off in a similar fashion but changes drastically when it gets to the regret. Also there is no Learning rate parameter in the code.
A "weakness" of NormalHedge is that it is not intuitive why the algorithm works. There a few proofs given in the paper by Chaudhurri et al. [4]. What they show is that NormalHedge has the same lower regret bound as SafeHedge, AdaHedge and FlipFlop. However, they do not really explain how the method was devised. This is why it is so interesting to test and compare NormalHedge with AdaHedge and FlipFlop which are the top alternatives. Nobody knows what will exactly happen.

Initially: Set $R_{i,0} = 0$, $p_{i,1} = 1/N$ for each i.
For $t = 1, 2, \ldots$

1. Each action i incurs loss $\ell_{i,t}$.
2. Learner incurs loss $\ell_{A,t} = \sum_{i=1}^{N} p_{i,t}\ell_{i,t}$.
3. Update cumulative regrets: $R_{i,t} = R_{i,t-1} + (\ell_{A,t} - \ell_{i,t})$ for each i.
4. Find $c_t > 0$ satisfying $\frac{1}{N}\sum_{i=1}^{N} \exp\left(\frac{([R_{i,t}]_+)^2}{2c_t}\right) = e$.
5. Update distribution for round $t+1$: $p_{i,t+1} \propto \frac{[R_{i,t}]_+}{c_t} \exp\left(\frac{([R_{i,t}]_+)^2}{2c_t}\right)$ for each i.

Figure 2: Summary of the NormalHedge algorithm [4]

4 Experiment Reproduction

To start the tests off, the experiments of Grünwald et al. are going to be replicated. If the results are the same, we have a good confirmation that our matlab code is correct. This is important because, even though we have the code for AdaHedge, FTL and FlipFlop, we only have the pseudo-code for NormalHedge and no code for the creation of the data sequences.

There are four experiments, which each show the strengths and weaknesses of the the algorithms. In all four the situations there are two experts and the data sequence of losses is $T = 1000$ long. These data sequences consist of an artificial initial loss vector l_1 followed by vectors $l_2, \ldots, l_T$ which are (0 1) or (1 0). The way that the next vector is chosen each round is by looking at the cumulative

loss difference of the two experts $L_{1,t} - L_{2,t}$. We then set a goal $f(t)$and chose the subsequent vector that brings the cumulative loss difference the closest to that goal. We plot the regret versus the time for the following algorithms:

1. Hedge with constant $\eta = 1$
2. Hedge with $\eta = \infty$ (Follow-The-Leader)
3. SafeHedge
4. AdaHedge
5. FlipFlop with parameters $\phi = 2.37$ and $\alpha = 1.243$ as in [4]
6. NormalHedge

4.1 Experiment 1 & 2: Best and worst-case for FTL

The first experiment that is going to be reproduced is the worst-case for FTL. Here we start of with $l_1 = (\frac{1}{2}\ 0)$ and $f(t) = 0$. This gives us the following loss sequence:

$$\begin{pmatrix} \frac{1}{2} & 0 & 1 & 0 & 1 & \dots \\ 0 & 1 & 0 & 1 & 0 & \dots \end{pmatrix}$$

We see that the results are the same in our test as in the original. FTL performs poorly as each round there is a different Leader. This results in an increase of 1 in the regret every two rounds. All the other algorithms also have the same results. One slight difference is the the original plots have a smoother lines. The probable reason for this is that the authors repeated the process a number of times and plotted them on top of each other or took the average.

The second experiment is the best case for FTL. Here we start of with $l_1 =(\ 0)$ and $f(t) = 0$ This gives us the following loss sequence:

$$\begin{pmatrix} 1 & 1 & 0 & 1 & 0 & \dots \\ 0 & 0 & 1 & 0 & 1 & \dots \end{pmatrix} \tag{1}$$

The reason this is the best case for FTL is that the leader never changes. FTL puts all its weight on the leader and that is always correct except in the first round when the weights are equally distributed. As we can see here, all the algorithms seem to perform like they should. The only exception is NormalHedge. In the original version it seems to oscillate between 0 and 1. After manually calculating what the values should be of the regret it seems like the original is mistaken. The reason for this is unclear. The only explanation we have is that the authors wanted to make clear that the FTL and FlipFlop lines are underneath the NormalHedge line and thus artificially made NormalHedge oscillate.

4.2 Experiment 3 & 4: Weights do not and do concentrate for AdaHedge

The last two experiments have $l_1 = (0\ 1)$ with $f(t) = t^{0.4}$ and $l_1 = (0\ 1)$ with $f(t) = t^{0.6}$ respectively. These are situations where AdaHedge works very well in the one and works very poorly in the other.
The data sequences created are too long to show but in the first situation the first expert has accumulated 508 loss, while the second expert has only 492. In the second and final experiment the gap is larger with the first expert having loss 532 and the second 468.

There are two things that are noticeable. The first is that the Hedge algorithms seem to work perfectly. This is not very surprising as our code is based on Grünwalds [3] but it is reassuring to see.
The second is that the NormalHedge line plot is giving slight different results. In both cases they do become constant but the value where they become constant and after how many steps that happens is a bit off.

5 Complex experiment

This final experiment, is similar to one described by L. Jansen [6], has been devised to try and make NormalHedge perform poorly in comparison to AdaHedge or FlipFlop. The loss data is made up as follows:

Let $y^T = (y_1, \ldots, y_T)$, be a random sample drawn i.i.d.from a Bernoulli($\frac{1}{2}$) distribution, with $y_i \in \{0, 1\}$. These values in y^T are the correct values each expert $k \in K$ will have to predict. Each round however, is either an easy round or a hard round. This is decided each round by a random draw from a Bernoulli(γ) distribution.
In an easy round, every expert makes a correct prediction so everybody's loss is zero. In a hard round every expert has probability λ_k of making a correct decision. If the expert makes an incorrect prediction the loss is 1, the prediction is considered completely wrong.

In our experiment we have two types of experts. There is one "good" expert and there are $K-1$ "bad" experts. The good expert has a higher probability of making the proper prediction in a "hard" situation. To make the good expert even better we make a prior distribution $\pi(K)$ of the experts. In all experiments we will have $\pi(1) = 0.5$ and $\pi(2) = \pi(3) = \cdots = \frac{1}{2(K-1)}$.
What this does is that the prediction of the good expert is given more weight. Say we have K experts, in the program, the results of the good expert are multiplied by (K-1). This results in there being (K-1) identical good experts and (K-1) bad experts.

What we want to achieve with this construction is that in theory there should be one clear best expert. But because we are going to work with such large amounts of experts, it is very likely that some bad experts will for short periods

of time perform better. This will result that relatively high weights are put on bad choices.
It will also cause there to be quite a few leader changes during the first part of the run. As seen in the first experiment in the previous section, NormalHedge does not perform very well in that case.

5.1 Experiment 1

In this first experiment we have chosen $\gamma = 0.7,\ \lambda_1 = 0.5$ and $\lambda_2 = \cdots = \lambda_k = 0.4$. We also set $T = 1000$ and $K = 1000$. This means that there are 1000 experts and that the Learner is going to predict for a 1000 rounds.

What we can see in Figure 9 is that NormalHedge and AdaHedge seem to perform almost exactly the same way. This is very surprising as we expected NormalHedge to perform poorly. We do not have an explanation for this phenomenon. At the start, the two algorithms differ a bit but they quickly converge. These two algorithms do both use previous data to make their prediction but how they do this is quite different as shown in the previous section.
What does look as expected is the data sequence. We can see that at the start there are a lot of switches between best experts which makes the regret of the algorithms increase rapidly. After a while, the good expert becomes so dominant though, that it is by far the best expert and never loses its top spot. This translates itself into the constant line of almost all the algorithms. The only one that doesn't even out until the very end is SafeHedge but that is exactly how SafeHedge is expected to behave.
We also see that FlipFlop performs badly. This is due to the fact that, of all the algorithms, FTL performs the worst , so when FlipFlop switches to FTL for a very short time, this increases its regret by a relativly high amount.

5.2 Experiment 2

For the second experiment, the values are: $\gamma = 0.7,\ \lambda_1 = 0.4$ and $\lambda_2 = \cdots = \lambda_k = 0.6$. So now we have that the good expert is worse than the bad experts but the distribution stays the same.

What we see is that FTL performs atrociously in comparison to the other algorithms. AdaHedge and NormalHedge perform exactly the same again so we cannot comment on one being better than the other. What we do see is that Hedge with a Learning rate of 1 performs surprisingly well. This could mean that the learning rate used by AdaHedge is also always around 1. By chance, $\eta = 1$ turns out to be a could choice. One can understand that testing for all constant Learning rates is not an efficient tactic though and that a dynamic strategy is still preferable.

5.3 Experiment 3

In the last experiment we wanted to see what a change in the γ would result in. We kept the value of the second experiment for the lambdas and we changed the value for gamma to 0.9. What we expect is that there will be less leader changes as in 90% of the cases, all the experts will be correct due to the rounds being easy.

When looking at the plot we see that as expected there are fewer leader changes which make it a good data sequence for FTL. We see that that algorithm performs the best. We also note that the overal regret is much higher than in the previous examples. This could be due to the fact that there are less leader changes, the algorithms have less opportunities to perform better than the best expert so far.
Lastly, we see that this is the first data sequence where FlipFlop performs better than NormalHedge and AdaHedge (slightly). This, like in the first experiment, is attributed to its ability to use the FTL strategy. We could see this data sequence as a relatively easy one, which also explains why SafeHedge performs the worst of all. The η used is too pessimistic and strict.

5.4 Conclusion

Sadly the goal of this paper has not been achieved. We did not find clear weaknesses in the NormalHedge algorithm in comparison to AdaHedge and FlipFlop. We were not able to break it. There are two reasons why this could be the case.
First, the code used for the NormalHedge algorithm could be flawed. In the simple experiments we saw that our results for that algorithm did not always match Grünwald et al's. In the case of the second experiment we are fairly convinced that our results are correct but in the last two we do not know which one is right. A comparison of the codes would be useful or the code written by the creators of NormalHedge could be requested.
The second is that NormalHedge just is a good alternative to AdaHedge and FlipFlop and that in most cases it will perform well. In all the cases we tried, which are numerous, NormalHedge and AdaHedge performed almost exactly the same. This is the more positive outlook.
Having said this, we cannot conclude that the Algorithm will do well in "all" cases. There are still numerous types of problems that can be tested and there is definitely room for further research and experimentation.

What did come out of this experiment is that we found a situation that, by changing just three values, can create data sequences that show the strengths and weaknesses of all the algorithms. They display the key properties of the algorithms. For example the fact that SafeHedge regret still increases while the others are constant or that FTL regret never decreases. Lastly, these sequences illustrate nicely the workings of FlipFlop and its dependence on AdaHedge and

FTL. This can give us a better understanding of its workings and could help to improve it.

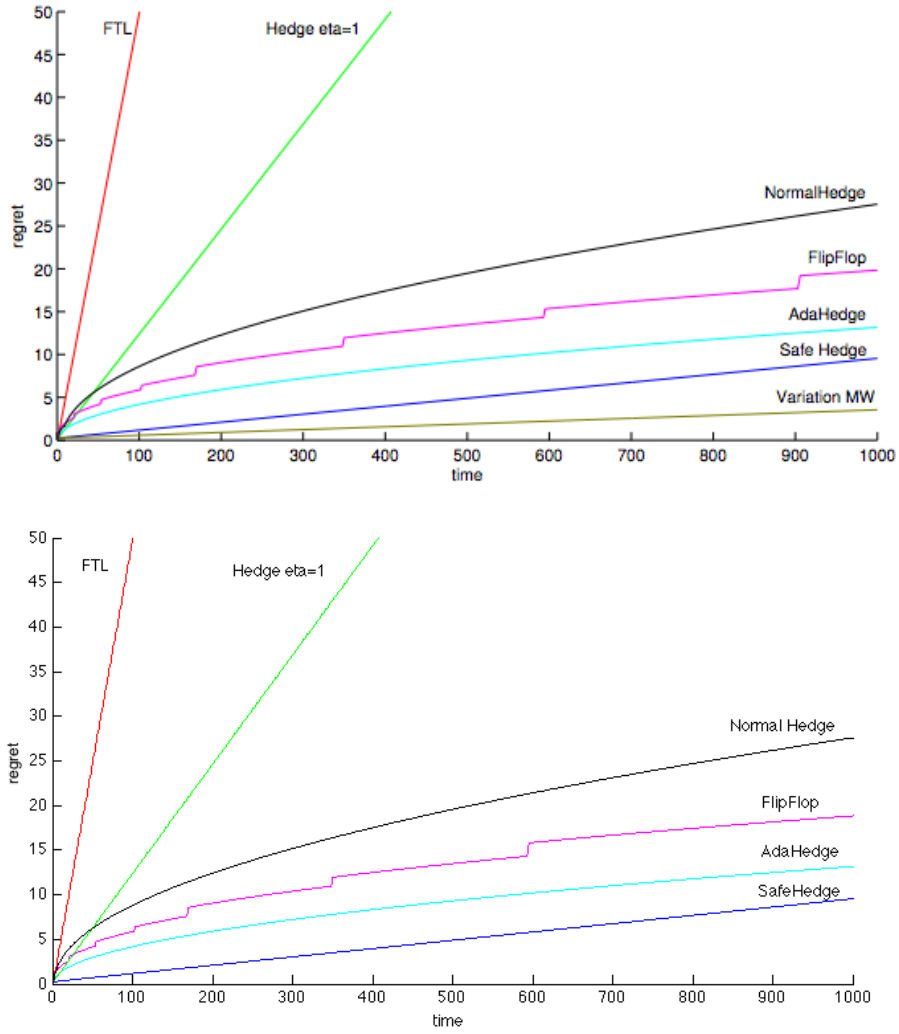

Figure 3: Comparison between Grünwald's (top) and reproduced experiment (bottom) for FTL worst-case

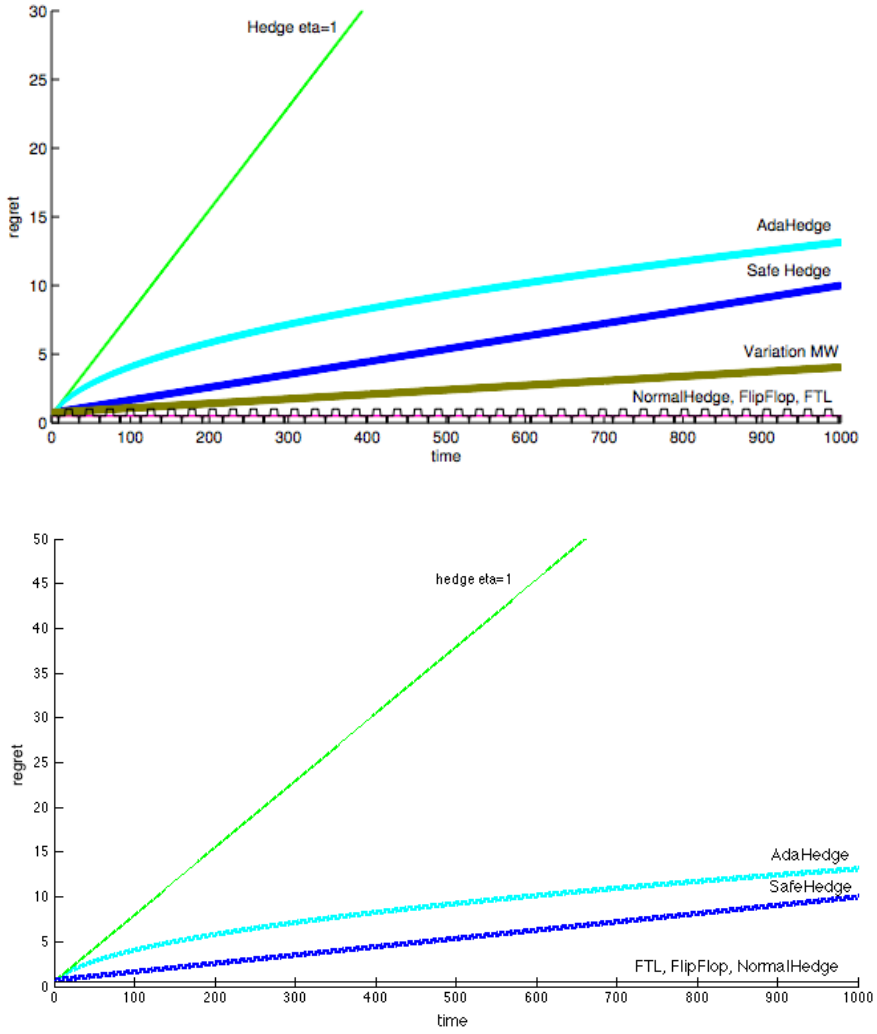

Figure 4: Comparison between Grünwald's (top) and reproduced experiment (bottom) for FTL best case

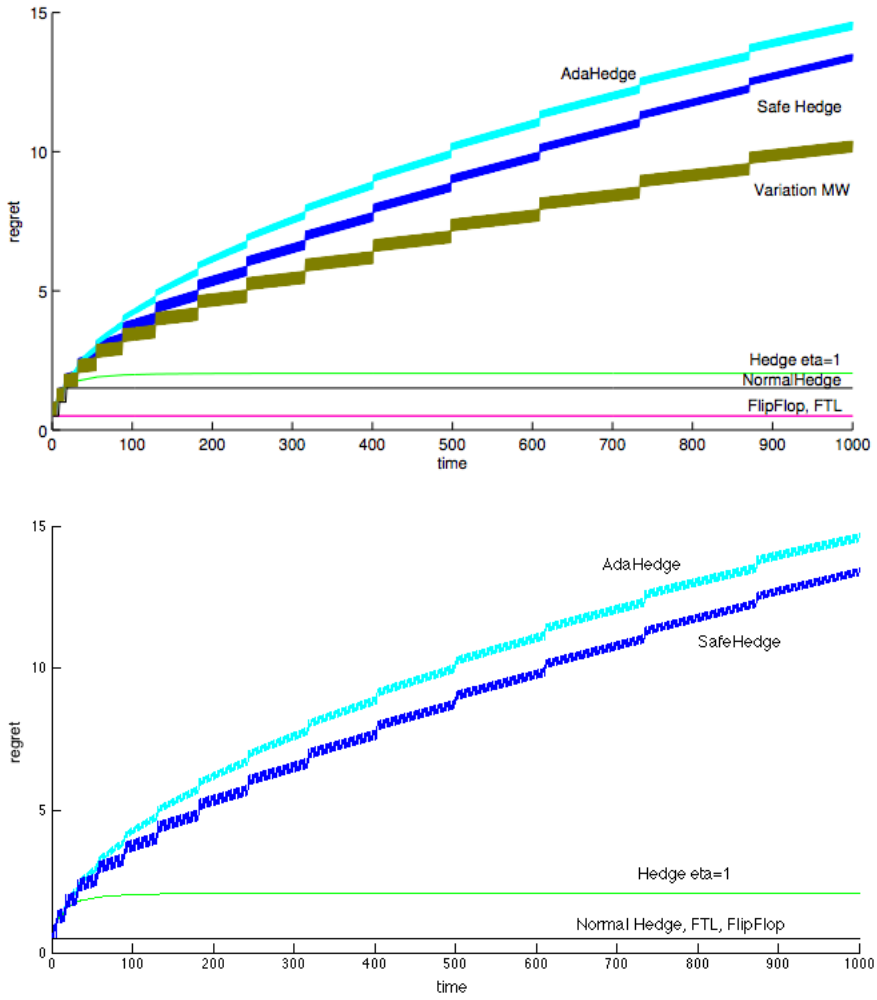

Figure 5: Comparison between Grünwald's (top) and reproduced experiment (bottom) for AdaHedge worst-case

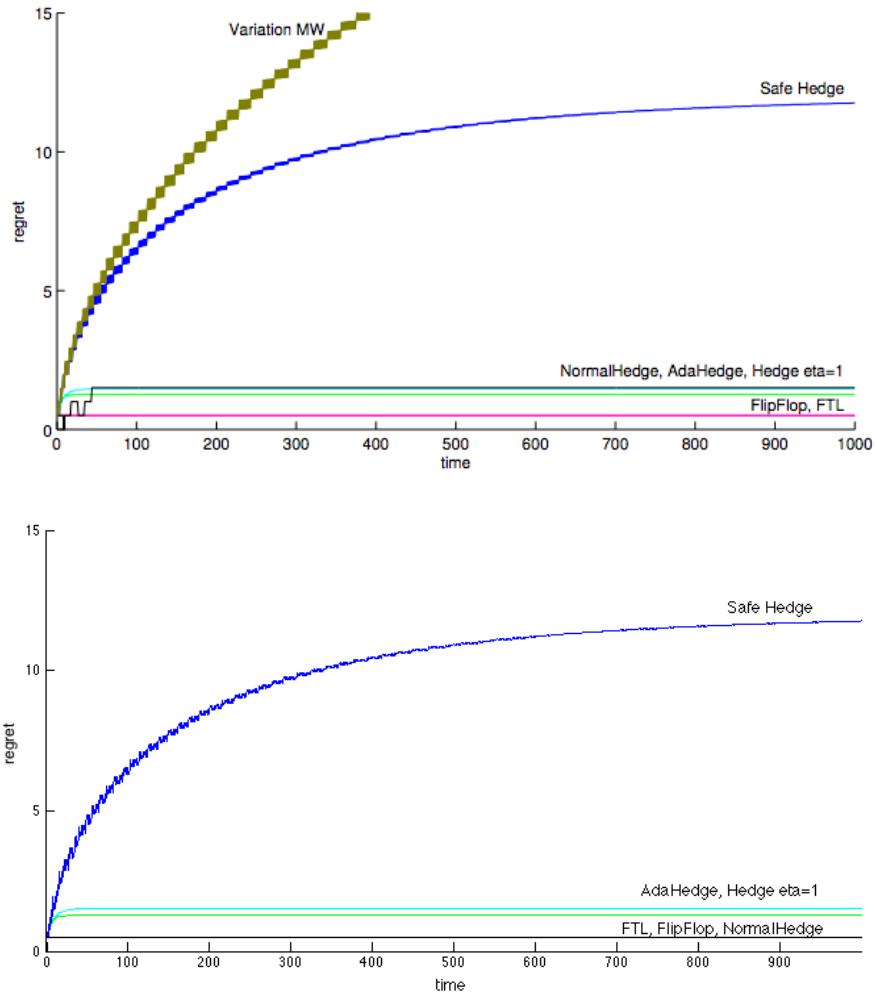

Figure 6: Comparison between Grünwald's (top) and reproduced experiment (bottom) for AdaHedge best case

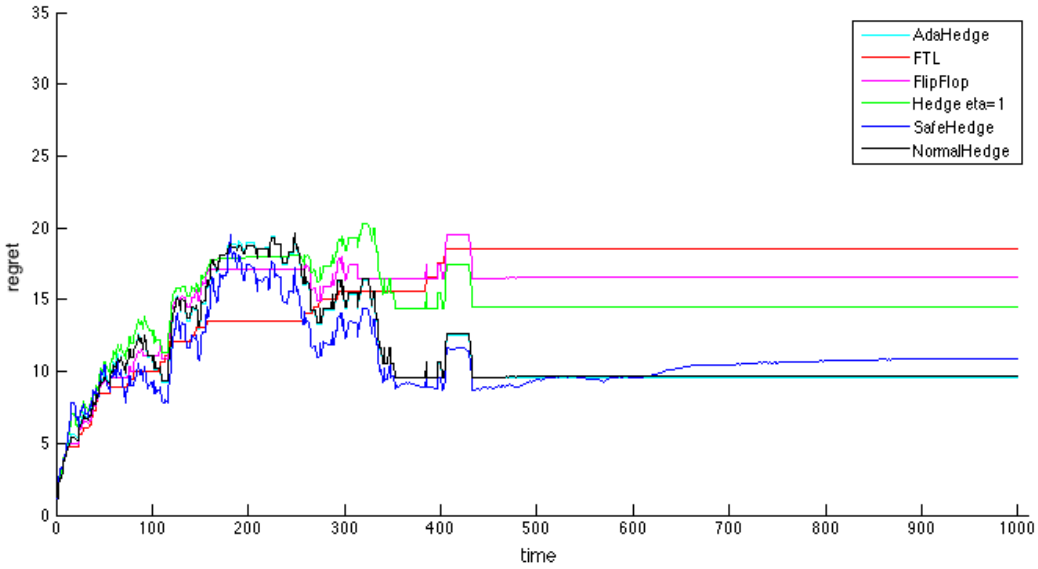

Figure 7: Plot of experiment 1 of all algorithms using the complex data sequence

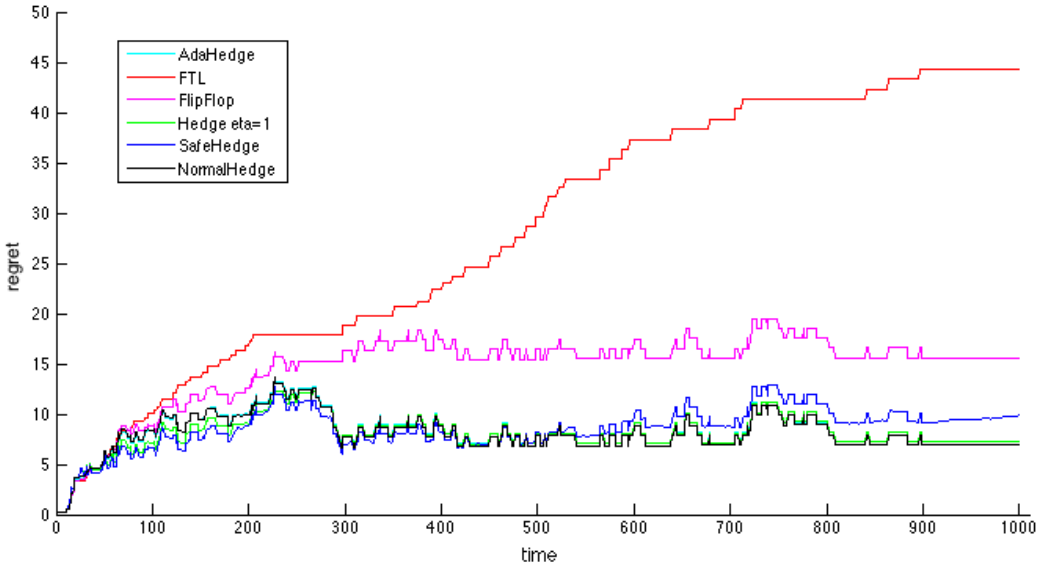

Figure 8: Plot of experiment 2 of all algorithms using the complex data sequence

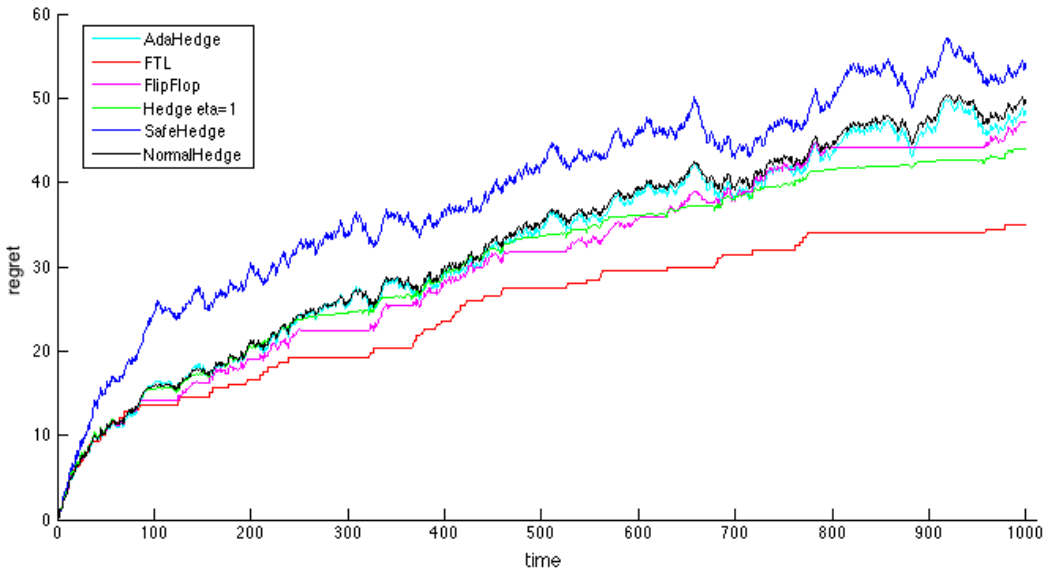

Figure 9: Plot of experiment 3 of all algorithms using the complex data sequence

6 Matlab code

Here is an overview of all the matlab code used to produce the results shown in the previous two sections.

AdaHedge and constant η code
This AdaHedge algorithm uses the same principle as the basic Hedge code in Figure 1 but has a much more complicated code to find η. This is partially done with the *mix* code described on the right of the Figure. To transform this into a constant learning rate algorithm one just needs to change the learning rate in the code to the desired value. For FTL, for example, you have to change η to an extremely large number (preferably the programable equivalent of infinity).

```
% Returns the losses of AdaHedge.
% l(t,k) is the loss of expert k at time t
function h = adahedge(l)
    [T, K]  = size(l);
    h       = nan(T,1);
    L       = zeros(1,K);
    Delta   = 0;

    for t = 1:T
        eta = log(K)/Delta;
        [w, Mprev] = mix(eta, L);
        h(t) = w * l(t,:)';
        L = L + l(t,:);
        [~, M] = mix(eta, L);
        delta = max(0, h(t)-(M-Mprev));
        % max clips numeric Jensen violation
        Delta = Delta + delta;
    end
end
```

```
% Returns the posterior weights and mix loss
% for learning rate eta and cumulative loss
% vector L, avoiding numerical instability.
function [w, M] = mix(eta, L)
    mn = min(L);
    if (eta == Inf) % Limit behaviour: FTL
        w = L==mn;
    else
        w = exp(-eta .* (L-mn));
    end
    s = sum(w);
    w = w / s;
    M = mn - log(s/length(L))/eta;
end
```

Figure 10: Matlab code for the AdaHedge algorithm [3]

FlipFlop code
This code is similar to that of AdaHedge. It uses the same *mix* code described in Figure 10 and in bold are the new elements.

```
% Returns the losses of FlipFlop
% l(t,k) is the loss of expert k at time t; phi > 1 and alpha > 0 are parameters
function h = flipflop(l, alpha, phi)
    [T, K]  = size(l);
    h       = nan(T,1);
    L       = zeros(1,K);
    Delta   = [0 0];
    scale   = [phi/alpha alpha];
    regime = 1; % 1=FTL, 2=AH

    for t = 1:T
        if regime==1, eta = Inf; else eta = log(K)/Delta(2); end
        [w, Mprev] = mix(eta, L);
        h(t) = w * l(t,:)';
        L = L + l(t,:);
        [~, M] = mix(eta, L);
        delta = max(0, h(t)-(M-Mprev));
        Delta(regime) = Delta(regime) + delta;
        if Delta(regime) > scale(regime) * Delta(3-regime)
          regime = 3-regime;
        end
    end
end
```

Figure 11: Matlab code for theFlipFlop algorithm [3]

NormalHedge

```
% This code returns the NormalHedge losses
function LA = normalhedge(l)
[T, N] = size(l);
R = nan(T+1,N);
R(1,:) = 0;
W = nan(T+1,N);
W(1,:) = 1/N;
LA = nan(T,1);

for t = 1:1:T
        LA(t) =  sum(W(t,:).*l(t,:));% Loss learner
        R(t+1,:) = R(t,:) + (LA(t) - l(t,:));
        R1 = R;
        R1(R1<0) = 0 ;% Al negative R values become 0
        ct = find2(R1(t+1,:),N);

        W(t+1,:) = (R1(t+1,:)./ct).*exp(R1(t+1,:).^2./(2*ct)); %
        tot = sum(W(t+1,:));
        W(t+1,:) = W(t+1,:)/tot; % normalisation

end% for
end% function
```

```
% Binary Search for ct
function ct = find2(R,N)

cmin = 0;
cmax = 1000000;
ctest = 0;
LHS = 0;
while abs(cmax - cmin) > 10^-4 % error interval

        ctest = (cmin+cmax) / 2;
        LHS = (1/N)*sum(exp((R.^2/(2*ctest))));

                if LHS < exp(1)
                        cmax = ctest;
                else
                        cmin = ctest;
                end
end% while

ct = ctest;
end% function
```

7 Bibliography

References

[1] Y. Freund and R. E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119139, 1997.

[2] V.Vovk. A game of prediction with expert advice. *Journal of Computer and System Sciences*, 56(2):153173, 1998.

[3] S. de Rooij and T. van Erven and P.D. Grnwald and W. Koolen. Follow the Leader if You Can, Hedge if You Must. *Journal of Machine Learning Research* 15, pages 1281-1316, April 2014.

[4] K. Chaudhuri, Y.Freund, and D. Hsu. A parameter-free hedging algorithm. In *Advances in Neural Information Processing Systems* 22 (NIPS 2009), pages 297305, 2009.

[5] N. Cesa-Bianchi and G. Lugosi. Prediction, Learning and Games. *Cambridge University Press*, 2006.

[6] L. Jansen. Master Thesis: Robust Bayesian inference under model misspecification. *Mathematical Institute Leiden*, July 2013

[7] A. Biaggi. Combining the normal hedge algorithm with weighted trees for predicting binary sequences. UC San Diego: b6700116, 2010

[8] N. Cesa-Bianchi, Y. Mansour, and G. Stoltz. Improved second-order bounds for prediction with expert advice. *Machine Learning*, 66(2/3):321352, 2007.