Incremental Tree Induction

Nicholas Piël

2007-11-21

Abstract

Analyzing data streams has in the recent years gained significant interest. This is mainly powered by the need to be able to classify over a continuos flow of data, for example for credit card fraud detection or human behaviour monitoring. This requires a high speed algorithm in order to be able to keep up with the data. While this is already a hard task, the task becomes more challenging since the underlying logic that generates the stream of data usually isn't static. For example people their behaviour might change over times, such flexibility in the data is know as 'concept drifting'. In this article i describe a technique i have implemented that tackles both challenge successfully. The technique works on the basis of a Genetic Algorithm, this is remarkable since GAs are commonly thought of to be slow.

1 Introduction

For the creation of decision trees from a set of data there are already lots of robust algorithms which have a proven performance. They are not only known for their high accuracy but also because the creation of the tree is a transparent process and the end result is a scheme which is easy to be interpreted even for non technical users.

There are two common approaches to the induction of decision trees. Those based on some greedy search by using scoring metrics such as information gain and gain ratio, such as with C4.5 [5] and the induction of trees by applying genetic algorithms. The first approach is known for its performance and its construction of trees has been proven to be NP-complete. The later approach while far from new has only fairly recently been tested extensively and compared with C4.5 [4]

These algorithms operate under the assumption that all trainingdata is available at a single point in time. And that the datapoints within the dataset are time independent. These assumptions should make it clear that there are already two situations which require a different approach.

1. When data becomes available sequentially and we need to continuously classify incoming data. This will require a high speed algorithm
2. When the rules that generate the data change over time we need our classifier to be flexible and adapt to the incoming data. This requires our algorithm to be robust under concept drifting.

The need for a fast algorithm seems to be understood by the field, for example Domingos has developed a Very Fast Decision Tree Learner called VFDT [1] and a Concept-adapting addition to this called CVFDT [3] which is able to cope with concept drifting.

In this article i will describe an implementation of a classifier targeted at a continuos stream of fast paced concept drifting data. The algorithm is based upon a simple Genetic Algorithm (GA) [2], while i do prefer not to make use of this evolutionary metaphor it does however quickly summarises the algorithm. But since I do not make use of crossovers or model anything as a mirror of the real world, we could also call it an heuristic beam search over a set of candidates (the pool) where the best one is selected by some scoring (fitness) function and new candidate are added to this set by a random search from the best performing candidate (mutation). However, since this is a rather mouth full i will refer to these algorithms as 'GAs'.

In order to be able to evaluate the implemented algorithms I needed a dataset to test the performance. I have synthetically generated this data around a simple guessing game which i will explain in the next section. After that i will explain the approach i have taken to implement the GAs and will end with the results, a discussion and possible further improvements.

2 The Oracle Game

The reason that i have chosen to generate the dataset is because i wanted a clean environment to be able to test the performance and this data had to obey the characteristics to which our algorithm was targeted, namely the vastness of it and the concept drifting.

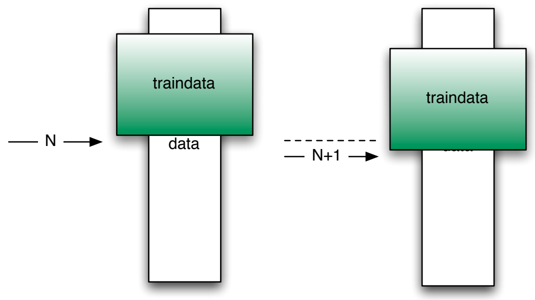
The dataset is based around some sort of game, which i will call the Oracle game. In this game there are a various amount of oracles O which know how to classify a real number between 0.0 and 1.0 as one of the categories C. It is simply a guessing game in which an unlimited amount of numbers are being randomly generated between 0.0 and 1.0 after each generated number the player (or in this case classifier) will have to guess to which category the number belongs. After each guess the oracle will let the player know if his guess is correct by proclaiming the real category. However, after a certain amount of generated numbers the oracle announcing the real categories may change including the rules it is living by. This is completely hidden from the player, unaware of the oracle and the rules it uses to announce the categories.

For example assume there are two oracles, Delpi and Pythia. And two classes: A, B. Delpi categorises every number as belonging to class A but Pythia categorises numbers between 0.0 and 0.5 as A and numbers between 0.5 and 1.0 as B. Now when an observer observes the following sequence: (0.04, A), (0.93, A), (0.43, A), (0.22, A) and is confronted with the following number (0.88, ?) his best bet would be to respond 'A'. Now what if after a while the sequence comes in as (0.6, B), (0.1, A), (0.8, B), (0.8, ?), what should he answer?

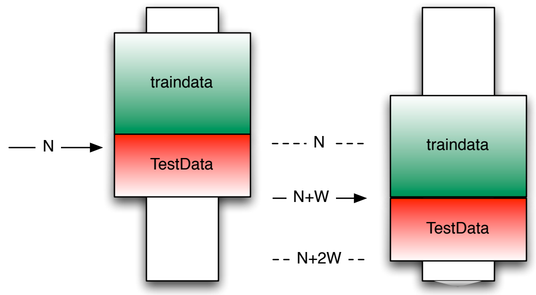
I have created a small program that will generate an endless amount of random numbers and will classify these by a certain number of oracles ($|O|$) from a group of classes C where each oracle classifies x numbers.

3 Incremental Induction

For the incremental induction of decision trees a simple approach would be to make use of a sliding window and have that iterate over the data. This can be done in two different ways, in the "Incremental Sliding Window" for each new datapoint we set the window up to the last N-1 datapoints and learn a classifier from that as can bee seen in figure 1(a). In the "Boxed Sliding Window" have a window for both the training data and the testdata and only slide these when the cursor runs out of the test window an example of this approach is given in figure 1(b).



(a) Incremental sliding window



(b) Boxed Sliding window

Figure 1: Different approaches to an incremental tree learner.

One would expect that the Incremental Sliding Window will outperform the Boxed Sliding Window. However this will come at a cost. It will require w times more calculations then the boxed classifier, where w is the size of the test set. This conflicts with the requirement that the algorithm has to have a good performance.

Another thing what this already shows is that the size of the test and training window already becomes a parameter which can influence the performance of the algorithm both in accuracy and time costs.

3.1 Approach

In order to be robust under the drifting of concepts and thus in our case the changing of Oracles. We must invent an algorithm which does not rely on the manual tweaking of parameters to match the data but an algorithm that adapts itself to the data.

In order for the algorithm to be able to track all changes in the data it needs some way to store this information. For an incremental tree learner that operates on an endless stream of data this can be done by keeping a fixed size cache (for example the traindata window as shown above) or storing all information in the tree itself.

For my solution i made use of some sort combination of both. The solution i came up with is some sort of evolutionary algorithm which consists of a pool of trees in which each tree keeps track of its accuracy on the last w items. Then, when a certain number of items have been classified by the best tree in the pool, the least performing trees will be removed from the pool and the best trees get mutated.

The idea is that the pool will converge to the incoming data in order to improve its accuracy. But also that trees that perform above par for a certain amount of time will not get killed immediately if their accuracy drops. The rationale behind this is that when we have a tree in our pool that for example classifies the Delpi oracle with an accuracy of 1.0 we do not want to immediately remove it from the pool if Pythia starts speaking and its accuracy drops. No, we want to keep it around so that if Delpi comes back our old tree can be picked up again and continue where it left.

3.2 Tree Mutations

In order to be able to do such mutations we need to think about how we are going to represent the trees and what the result of such manipulations should be. I represented the decision tree as a binary tree. This tree consists of non terminal nodes (points) over which data can be followed to end up in a terminal node (categories). Therefore our algorithm needs methods in order to adjust these points and categories, we can think of the following mutations.

1. Adding of node-points, to create a longer path or one with more branches
2. Deletion of node-points, to create a shorter path or a path with less branches
3. Changing the result of a certain path, thus the classification
4. Shifting the values of the node points in a path

I will show that these four possible mutations to a tree can be summarised by only two actions, namely insertion and deletion of a node. However, such mutation on a tree is not a single action but requires some extra effort in order to make sure the tree stays consistent and will not contain any redundant information. As redundant information will greatly slow down the performance of the algorithm.

In the coming subsections i will detail the various mutations and the consequences of each.

3.2.1 Node Insertion

The insertion of a node can lead to three different results, the first is that a new node added to the tree bears no information thus the result is essentially the same tree. For example, lets say we have 3 classes distributed in some arbitrary way within the range 0.0 - 1.0. When the initial tree is defined as [(0.0, A);

(0.5, B)] meaning all numbers from 0.0 to 0.5 belong to category A, and all numbers larger then 0.5 belong to B. When we try to add the following node (0.75, B) meaning all numbers larger then 0.75 belong to class B then this will add no further information to the tree. And thus the tree stays the same without modification.

The second result of a node insertion is that the point of a node is being modified. For example when we take the same initial tree as above. And add the following information: (0.25,) thus meaning everything larger then 0.25 is B. We only have to adjust the value point in the initial tree from 0.5 to 0.25 to obtain the same result.

The last result can be observed when want to add the node (0.75, C) into the same tree, the only way to represent this is to add a new node.

The algorithm for node insertion is outlined like this:

1. Given the node and a category c walk over the tree starting from root until we end up in a pre terminal node
2. If c equals one of the child terminal nodes their categories mutate point
3. Apply look-ahead, if the first next node (requires tree traversal) equals the current class then mutate point (set it to the current value).
4. Otherwise replace terminal class with new point and attach the replaced terminal and the new class to this new node.

3.2.2 Node Removal

The removal of a node can lead roughly to the same kind of results as the insertion. When we 'delete' a node we start with a certain value at the root node and search for the matching terminal node down the tree. Once we reach the terminal node we will set its class to one of the possible classes. If the new class is equal to the current class we simply ignore the mutation, otherwise we will simply mutate the class and clean the tree where needed.

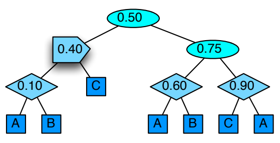
The algorithm for node removal is somewhat like this:

1. Select a random path until we reach a terminal node
2. Create the new value for the node
3. If equal to current node Ignore
4. If equal to sibiling then set parent node to terminal and adjust value of parent.parent
5. If next right node is equal then set common parrent to new value and replace parent with sibling
6. If next left node is equal then set common parrent to new value and replace parent with sibling

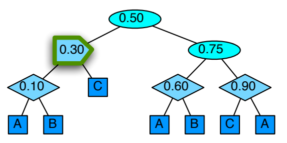
The result of this is that mutations on the tree can be done with a complexity of O(log n), where n is the number of nodes within the tree. It also keeps the tree neat and ordered and thereby limiting the amount of nodes within the tree.

3.3 Learning Algorithm

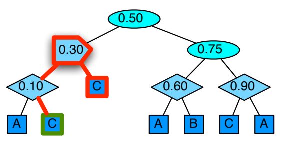
The algorithm itself is based on an evolutionary algorithm which consists of a pool which contains a set of trees. And then the pool is trained on some sliding window of items, then the best classifying tree from this pool is picked and will be used to classify items on the test set. The way in which the training window and test window is set differs, and I have chosen two different approaches for them. Both approaches however start with a pool that has been trained on an initial data set.



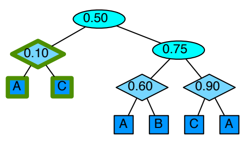
(a) The initial tree



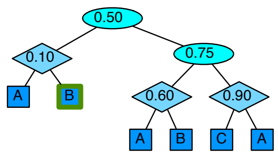
(b) Inserting (C, 0.3) leads to a mutation



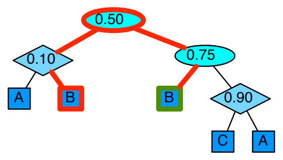
(c) When deleting at 0.10 and replacing with C leads to a resolved conflict with C at 0.30



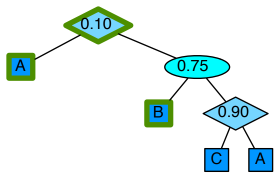
(d) The conflict has been resolved



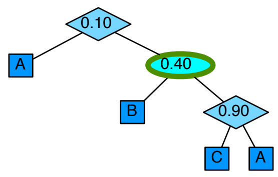
(e) Deleting at 0.10 and replacing with B leads to a simple mutation



(f) Deleting at 0.55 and replacing with B leads to a double conflict. The situation is shown after the conflict with the sibling has been resolved



(g) And now the constraint with the 'left next node' has been resolved



(h) Inserting C at 0.40 leads to a mutation

Figure 2: Mutations on the tree

3.3.1 Slidingbox Incremental Learner

In the first approach the algorithm operates with a sliding window over the data. The pool is being trained on the trainingwindow, then the best classifier is picked out of this pool and used to classify the test window. When the end of the test window has been reached, the slidingwindow gets shifted and the process is repeated all over again.

For the training we use a really simple iterative algorithm. In each iteration all trees in the pool are being tested on the data in the training set. They are then being sorted on accuracy after which the bottom half gets removed and the top half is allowed to mutate.

The initial training of the pool operates in the same manner as explained above, but the number of iterations is normally higher since there are no restrictions to time cost of this training as it is offline. An example of the accuracy of the best-tree in the pool and the average accuracy of all trees in the pool over a certain amount of iterations is depicted in figure 4.

3.3.2 Health Point Classifier

In the 'sliding box' approach we still needed to retrain the pool once in a while. While this can be done relatively fast as we will see later on, it is still a somewhat costly method. The 'health point classifier' does not have this repetitive learning every once in a while but continuously learns from each incoming datapoint.

We start with a pool trained with the previous method, we also still make use of some sort of window on which we are going to measure the performance of our classifiers. However where in the previous case the performance of each tree was only calculated at the end of each window we constantly update the accuracy of each tree and store this information in the tree itself.

Then when a new point arrives, we select from the pool the tree that had the best accuracy over its window to classify the next incoming point. Next, we assign some sort of 'health points' to all trees corresponding with their observed accuracy. We also substract 1 health point from each tree for each datapoint that enters. The net result of this is that better performing trees will increase health points and those that perform less will loose health points. Also, since the amount of health points each tree gets is simply a ratio of its accuracy compared to the sum of accuracies of all trees, a tree or hypothesis will be rewarded more points

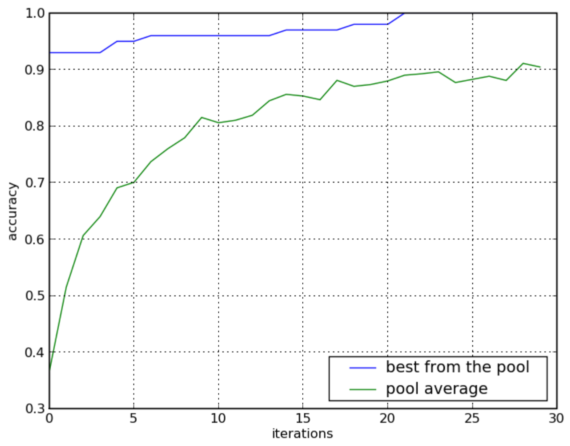


Figure 3: Shows the training of the pool over different iterations. From a pool with size 100 we can see that it already contains a tree that has an accuracy over 0.9. After a few iterations the avarage accuracy of the complete pool went up from 0.35 to 0.9

if its performance is more above par.

When the health points of a tree reach zero and there is no chance that it will outperform the current best performing tree in the current window, the tree is removed from the pool and will be replaced with a mutation on the best tree. While we haven't really removed the window parameter from the algorithm itself we do have restricted its influence as now a smaller window will not cost us extra calculation time.

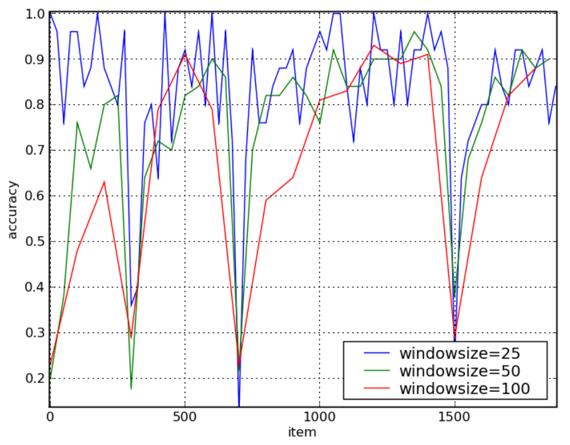


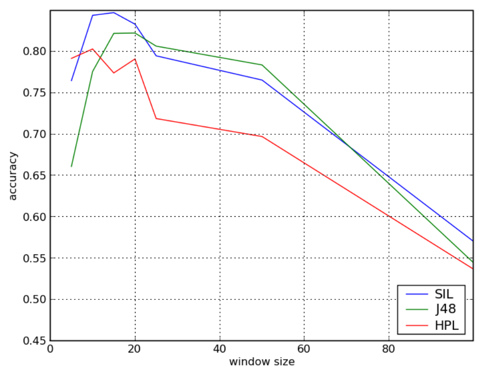
Figure 4: Showing the accuracy when we set different windows for the HP classifier

In figure 4 we can see the influence of 3 different window sizes, the larger the window the more it relies on older data which does not represent the problem currently at hand. This indicates that smaller windows are expected to perform better.

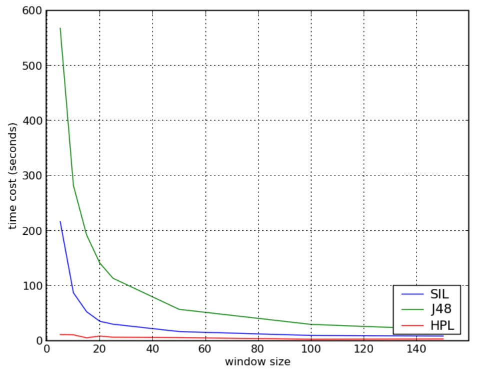
4 Results

All our results where obtained on data generated by our oracles, the base settings where 20 different oracles each having a quota of 100 items and 3 different categories. With this amount of oracles the data becomes really hard to classify by one of the standard approaches. For example when we take J48 (the C4.5 implementation in Weka) as the baseline and have it build its model on the generated dataset and then let it classify the same data again with the tree it generated, it can be easily shown that our learners will greatly outperform this method.

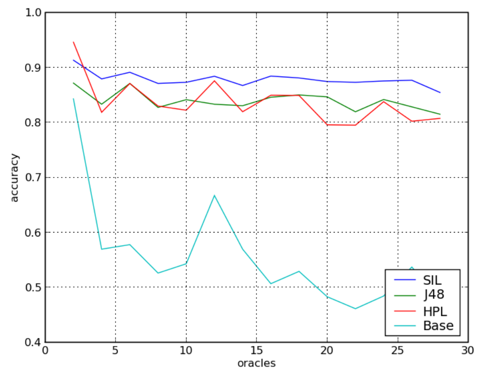
Such an comparison would not be fair, as J48 is unaware of the dependancy between the sequence of items. To even this out i have included a different approach based on the J48 classifier in which it trains and tests on the same sliding window as the 'sliding iterative learner' does. In the coming sections i will refer to this sliding box approach of J48 as just 'J48', I will refer to the regular J48 which operates on the complete dataset all-at-once as 'base'.



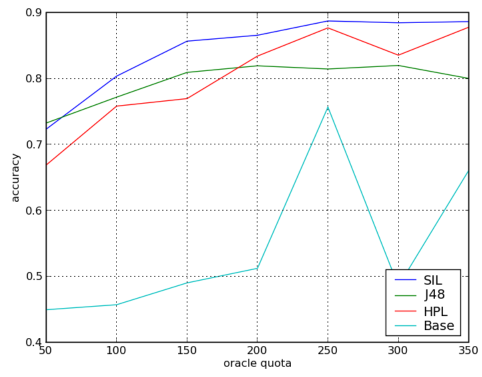
(a) Accuracy over different window sizes



(b) Cost in time of different window sizes



(c) Accuracy over different different amount of oracles



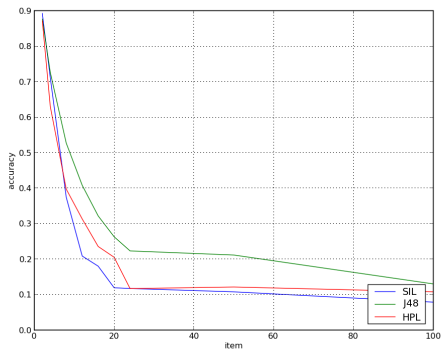
(d) Accuracy over different quota for each of the 20 oracles

Figure 5: Performance over different settings

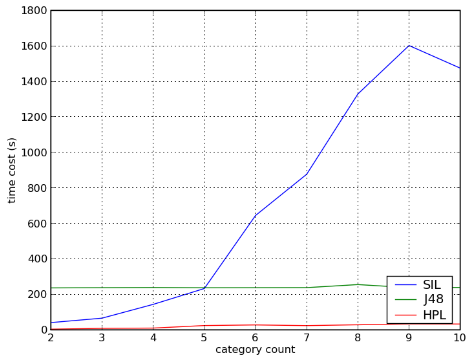
In figure 5 we can see the performance over different window sizes in both accuracy and time cost. In figure 5(a) we can see that the accuracy of smaller window sizes generally outperform the larger ones. We can also see that for the sliding window approaches (thus SIL and J48) tempt to overfit to the data if the window gets too small. The HP learner feels less of this effect and even from the sliding window approaches SIL shows less performance decrease then J48. The reason behind this will be that both approaches also store information about the data in the pool itself.

Figure 5(d) shows the timecost of all three approaches over the different windows sizes. Both sliding window approaches show a cost in time that grows exponentially when we decrease the window size. The Health Point learner stays constant in running size.

The experiments above show the results on different flexibility of the data. Ie much do the concepts drift. They do not show the result when the complexity of the problem itself changes. It will be interesting to see how the algorithm will perform on some harder problems. For this I tested the performance of the three algorithms on problems with different complexity by increasing the number of possible classes over which the algorithm had to classify.



(a) Accuracy over different classes



(b) Time cost over different classes

Figure 6: Accuracy over the complexity of the problem

From figure 6(a) it is interesting to note that the accuracy drops when the complexity of the problem increases, the reason behind this is that we test with the same setting as used above. An oracle quota of 100 items and a total of 3000 items. This seems to be too less to identify the problem for the classifiers. Experiments with the HP classifier showed that increasing the quota for the oracles and total item count will bring back the accuracy, and I did not even need to increase the window size to reach this high accuracy again.

Another side effect from the increasing complexity can be seen in figure 6(b). We can see that the running time for 'SIL' increases linearly with the complexity of the problem, where the time cost for 'J48' stays constant and the time cost for the 'HP' classifier increases only very light.

5 Discussion

The difference between 'J48' and 'SIL' can be explained by the influence of the window parameter. I have no doubts that J48 can easily perform as good the SIL learner when we set the window size accordingly to the oracle quota.

The window size seems to be a very important parameter in our approach, and as we have explained in our introduction we wanted to limit the need for such parameters as we have no way to establish the correct setting beforehand in a real live situation. While this might be the case for more static learners such as J48 on a sliding window. Initial experiments seem to show that there is no need to increase the window size even for very large oracle quota when we use the 'HP' learner.

Not only is the time cost of the 'HP' learner low, it also shows excellent performance compared with the other two methods. The performance difference that does exist between 'SIL' and the 'HP' learner can be explained by the simple case that HP converges slower to the optimal result. However, the results that when we give the 'HP' learner enough time (by increasing the Oracle Quota) the performance will not only surpass 'J48' but approach the 'SIL' learner.

The goal was to create an algorithm that would induct decision trees incrementally. For this i defined subgoals that the algorithm had to perform well measured in time cost and had to be able to adapt to flexible problems. I have shown that an evolutionary algorithm seems to be suited for exactly such an approach.

Not only does it adapt to concept drifting, but it can also be seen that the timecost of such an approach seems to be very low compared to alternative methods such as J48 over a sliding window. Such statements as this should always be taken with a grain of salt as the time cost of an algorithm depends heavily on its implementation. However, in this case I had shown that where the performance of algorithm that uses some sort of sliding window its time cost grows exponentially when we decrease the window size. Or in other words, such algorithms cost more time for more flexible problems. The time performance of the HP classifier stays constant.

While these initial results are very promising we should warn ourselves against over excitement. The algorithm was currently tested only on a single feature. While it will not be a problem to have it handle multiple features it could very well be that multiple features have a significant impact on the time performance. At least for the SIL classifier the results as shown in figure 6(b) indicate that the running time will grow linearly. The cause behind this is that the algorithm will easily add extra nodes and thus depth to a tree where it is really not needed. Some heuristics to help the algorithm with making these choices could counter this. I will present some in the next section.

6 Further Improvements

The HP learner shows excellent performance. However, i think that the performance of this simple algorithm can be boosted further. First of all, a weakness of it is that it adapts relative slow to a new oracle change compared with SIL. We might be able to lower this reaction rate by changing how tree's will mutate.

In the current situation a tree mutates by just selecting randomly an action (insert/delete) and a point and class. The point and class could be chosen by some fitness function over all the nodes within the tree itself. Where it more eagerly will change nodes that classify incorrectly. This will help the algorithm prune or mutate just those branches of a tree that perform below par. Such an addition can be done without adding significant time cost to the complete run of the tree.

Another option to increase the performance of the GAs would be to have the algorithm combine equal trees. The thought behind this is something like this. Say the real problem states that all points below 0.5 should be classified as A and all above as B. Now, when we generate random trees, some might be [A, 0.4, B] other [A, 0.6, B] now both trees will have the same performance for the datapoints: (0.1, 0.7, 0.3). What the algorithm could do is combine the two trees to a single tree by averaging their point values. The result would thus be: [A, 0.5, B] or in other words the real situation. We could even opt to have the single tree have an amount of healthpoints equal to the sum of both. Such an addition could not only be a better estimate of the real problem but it will also free up room in the pool for other mutations. When we apply this possibility to merge the tree only for the best tree at this moment, it will not penalise the time performance of the algorithm.

References

[1] P. Domingos and G. Hulten. Mining high-speed data streams. *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Datamining*, 2000.

[2] DE Goldberg and JH Holland. Genetic algorithms and machine learning. *Machine Learning*, 1988.

[3] Geof Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, 2001.

[4] Athanassios Papagelis and Dimitrios Kalles. Breeding decision trees using evolutionary techniques. *Proceeding of the Eighteenth International Conference on Machine Learning*, 2001.

[5] R Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.