

Reinforcement Learning Based Decision Tree Induction over Data Streams with Concept Drifts

Christopher Blake

Leibniz University Hanover & L3S Research Center
Hanover, Germany
chriswblake@gmail.com

Eirini Ntoutsi

Leibniz University Hanover & L3S Research Center
Hanover, Germany
ntoutsi@l3s.de

Abstract— Traditional decision tree induction algorithms are greedy with locally-optimal decisions made at each node based on splitting criteria like information gain or Gini index. A reinforcement learning approach to decision tree building seems more suitable as it aims at maximizing the long-term return rather than optimizing a short-term goal.

In this paper, a reinforcement learning approach is used to train a Markov Decision Process (MDP), which enables the creation of a short and highly accurate decision tree. Moreover, the use of reinforcement learning naturally enables additional functionality such as learning under concept drifts, feature importance weighting, inclusion of new features and forgetting of obsolete ones as well as classification with incomplete data. To deal with concept drifts, a reset operation is proposed that allows for local re-learning of outdated parts of the tree. Preliminary experiments show that such an approach allows for better adaptation to concept drifts and changing feature spaces, while still producing a short and highly accurate decision tree.

Index Terms—decision trees, reinforcement learning, stream mining, concept drifts

I. INTRODUCTION

Decision trees (DT) are one of the most popular classification models for machine learning due to their simplicity and interpretability. As a result a large variety of methods have been proposed for both batch and stream learning scenarios. In batch learning approaches, the complete dataset is given as input to the algorithm and can be accessed multiple times to select the best tree among multiple alternative hypotheses. On the contrary, in stream mining approaches, access to the data is limited and moreover, the stream is potentially infinite; therefore, the algorithm has to decide with limited information on how to grow the tree. Moreover, in a data stream the characteristics of the underlying population might evolve with time, leading to changes in the concept to be learned, a phenomenon known as concept-drift. To deal with concept-drifts the learning models need to adapt to changes quickly and accurately.

The majority of DT induction algorithms for both batch and stream learning are greedy, i.e., when selecting a splitting attribute, they try to maximize short-term quality measures such as information gain or Gini-index rather than maximizing long-term return. Reinforcement learning (RL) methods enable a better approach when such a sequential decision making process is involved because they try to find an optimal sequence of actions (policy) instead of selecting the most greedy action at

each time point. The RL agent finds such a policy by learning an estimate of the value of the different states.

This work builds upon the RL-based DT induction approach of [3]. In their work, the authors propose a formulation of the DT induction problem via RL in an online setting which allows updating the model with new instances from the stream, but it does not explicitly target outdated information. To overcome this shortcoming, we introduce a local state reset mechanism that increases the capability of adaptation to underlying data changes. Moreover, to increase the interpretability of the derived model, we summarize the MDP policy into a simplified human-readable decision tree. Finally, an extensive evaluation on the effect of different data stream-related and model-related parameters is provided.

The rest of the paper is organized as follows: Related work is overviewed in Section II. The problem and RL-based formulation are presented in Section III. The RL-based DT induction approach is introduced in Section IV-A with discussion on efficiency and convergence in Section IV-B. Experimental results are presented in Section VI. Finally, Section VII with conclusions and outlook completes this work.

II. RELATED WORK

Decision trees (DTs) are one of the most popular classification models due to their simplicity and interpretability [5]. A decision tree can be mapped into a set of rules, each rule corresponding to a path in the tree, and therefore it is easily understood by the end-user. The majority of decision tree induction algorithms follow a top-down recursive approach starting from the root. Two key decisions for building a DT model are the attribute split criterion for selecting the best attribute for splitting and the termination criterion for deciding on when to stop further tree expansion. Traditional decision tree induction algorithms like ID3, C4.5 and CART are greedy as they try to expand the tree at each step by selecting the attribute with the best splitting quality such as information gain or Gini-index, following a hill-climbing local search approach in the hypothesis space. However, due to its greedy nature, such an approach might lead into a sub-optimal solution. Moreover, most of the DT algorithms are batch learning algorithms, and therefore require the complete training set as input to the algorithm; algorithms like ID3, C4.5 and CART [5] fall in this category.

Decision tree models are also a popular classification method for data streams. In contrast to the batch case, in a stream setting the dataset is not known in advance and it is potentially infinite, therefore one cannot wait to make a splitting decision based on the complete dataset. The Hoeffding Tree (HT) algorithm [2] overcomes this problem by using the Hoeffding bound to decide with certainty about the best attribute for splitting based on the thus far encountered instances from the stream. This algorithm is incremental, meaning the model is updated based on new instances from the stream. However, the model does not forget and therefore outdated information cannot be removed from the model, causing difficulties with concept drifts. The Concept-Adapting Hoeffding Tree [4] overcomes this problem by monitoring the performance of the different tree nodes and re-learning the ones that are no longer effective. The Adaptive Size Hoeffding Tree (ASHT) [1] tackles this problem by introducing a maximum tree size limit and resetting the tree whenever this limit is reached. This model is used as a base learner in an ensemble of different size trees, which therefore reset at different rates; small trees are reset more often and therefore adapt faster to concept drifts, in contrast to larger trees that typically survive longer before reaching the size limit.

Most of the DT induction approaches, including both batch and stream DT induction approaches, are greedy meaning that they try to maximize the short term goal of selecting the next best splitting attribute instead of maximizing the long-term return. The latter is the subject of Reinforcement Learning (RL) [7], where the goal of the agent is to learn a sequence of actions (policy) that maximizes the return. Thus far, only a few approaches exist that instead of selecting the next best splitting attribute, try to select a policy. The RL-based DT approach of [3], which belongs to this group, tries to learn a policy that achieves good classification performance while making few queries. Their formulation specifies a state space from combinations of feature-value pairs to represent the paths in different decision trees and an action space consisting of queries and reports. A feature query action retrieves information about the value of a particular feature and a report action predicts the label. Query actions always incur a negative reward, thereby encouraging a shorter tree. Report actions incur a positive or negative reward dependent on correct or incorrect classification respectively, thereby encouraging an accurate decision tree.

The proposed RL-based DT induction approach of this work extends the formulation of [3] to allow handling of changing feature spaces. In particular, it proposes a reset mechanism that resets a state based on its Gini impurity (Section IV-C). This reset mechanism allows for local re-learning as a reaction to concept drifts. Moreover, a faster simplified tree is enabled by finding the best groups of queries, removing the need to first filter for valid queries which requires individually checking all possible query options at a state (Section V).

III. PROBLEM DEFINITION AND FORMULATION

An evolving data stream of instances arriving over time is assumed. At every time step, $t = \{1, 2, 3, \dots\}$, an instance

$\vec{x}_t \in X$ with an associated label $y_t \in Y$ is provided to the learner, with X being the input space and Y being the output space. Clearly, there is no prior knowledge of the input or output space before data instances arrive and the dimensionality of the feature space may change with time. For example, over the course of the stream, previously unseen features and feature values may appear and known features may no longer occur. Moreover, the output space is also assumed to be dynamic, that is, new class labels might appear as new instances arrive or existing classes might vanish over the stream. Finally, the underlying stream population is subject to changes over time, leading to changes in the concept to be learned (concept-drifts).

The DT induction problem is formulated as a Markov Decision Process (MDP), building upon the formulation of [3]:

- **State** (s) A combination of features and their respective values known at that state.
- **State Space** (S) All possible states of the MDP.
- **Query Action** (\mathcal{F}) An action possibility at each state, which returns the value for a particular feature from the training input \vec{x}_t , and causes transition to another state. Naturally, a state cannot query a feature that it already contains.
- **Report Action** (\mathcal{R}) An action possibility at each state, which predicts the class label of the input instance.
- **Action Space** (A) All possible query and report actions.

Training occurs with the arrival of each new instance ($\vec{x}_t \in X, y_t \in Y$) at time step t from the stream by choosing between two actions: a *feature query action* \mathcal{F}_i and a *label report action* \mathcal{R}_j . The feature query action will introduce an additional feature into the policy, whereas the label report action will predict a class label $y' \in Y$ for \vec{x}_t . The prediction y' will be compared to the true class label y_t producing a reward and ending training. A policy is created or updated by propagating the reward from the reports through the queries. This leads to a policy that is both highly accurate and uses minimal information for classification [3].

Diverging from the formulation of [3], the reward system for queries and reports is modified. In particular, the reward for a report action \mathcal{R}_j is based on the percentage of an observed label at that state, and lies in the [0-1] range. A query action \mathcal{F}_i is only used to transfer reward between states using a discount factor γ and feature importance w_f as described below.

- 1) **Discount Factor** $\gamma \in [0, 1]$: The transfer rate of reward from a state's report reward and another state's query. It is usually between 0.8 and 0.99.
- 2) **Feature Importance** $w_f \in [-1, 1]$: An optional weighting to manually encourage (+1) or discourage (-1) feature inclusion during model training. It can also be used for offsetting class imbalance [3].

IV. RL-BASED DECISION TREE LEARNING

A. The Training Process

As a new instance ($\vec{x}_t \in X, y_t \in Y$) arrives from the stream at time point t , the training process proceeds as follows:

- 1) Start with the root state, which has no features but only query- and report-actions.
- 2) Identify queries with matching feature-value pairs to the current data instance.
- 3) Disqualify queries with lower expected reward than the classification label's (report's) expected reward.
- 4) If no queries have higher reward, then check the predicted label and go to step 5. Otherwise, go to step 6.
- 5) Adjust the expected rewards of the state's reports and end training for this instance.
- 6) Pick best query, usually by highest expected reward.
- 7) Perform the query and transition to the next state. Update the query's reward with a portion of the reward from the next state's report. Return to step 2.

1) *Action Decision*: At a current state s_n , the appropriate query or report action must be determined. A query-action will cause transition to a different state, whereas a report-action will end training for that instance, which updates the expected rewards of the reports at that state. Such a decision requires comparing the expected reward of all known query- and report-actions, selecting the action with the highest reward.

At any given state s_n , there are likely many query actions. A unique query at this state $\mathcal{F}_{s_n,i}$ exists for each combination of feature f , feature-value f_v , and classification label $y_j \in Y$ as shown in (1). However, for a given training instance many of these queries are not valid and must be filtered. Similarly, multiple report actions $\mathcal{R}_{s_n,j}$ may also exist at state s_n , one for each unique classification label y_j as shown in (2).

$$\mathcal{F}_{s_n,i} = \mathcal{F}(f_i, f_{v_i}, y_j) \quad (1)$$

$$\mathcal{R}_{s_n,j} = \mathcal{R}(y_j) \quad (2)$$

Because each query and report have an associated expected reward Q at that state s_n , the expected rewards are determined respectively using (3) and (4).

$$Q_{\mathcal{F}_{s_n,i}} = Q(\mathcal{F}_{s_n,i}) \quad (3)$$

$$Q_{\mathcal{R}_{s_n,j}} = Q(\mathcal{R}_{s_n,j}) \quad (4)$$

2) *Query Reward Calculation*: At a current state s_n , upon choosing a query $\mathcal{F}_{s_n,i}$ as the best action, the feature's value is obtained from the current instance \vec{x}_t . Using this additional feature-value pair, the next state s_{n+1} is found or created. If a new state is created, all new queries from this new state are given optimistic rewards of +1 to encourage exploration.

The expected reward of the next state $Q_{\mathcal{R}_{s_{n+1},j}}$ is retrieved and used to update the expected reward of the current state's query $\mathcal{F}_{s_n,i}$. The query's new expected reward is a function of the discount factor γ , feature importance w_f , and report reward $Q(\mathcal{R}_{s_{n+1},j})$ as shown in (5).

$$Q_{\mathcal{F}_{s_n,i}} = \gamma(1 + w_f)Q_{\mathcal{R}_{s_{n+1},j}} \quad (5)$$

If the feature's importance is set to -1, no reward is transferred. If the feature's importance is set to 0, only the discount factor is relevant. Therefore, new training data can deprecate a feature by setting the importance near to -1 or encourage inclusion of a feature by setting the importance near to +1.

3) *Report Reward Calculation*: Each report $\mathcal{R}_{s_n,j}$ for a particular class label y_j at state s_n is simply the percentage p_j of the thus far observed instances of the particular class at that state and lies in the $[0 - 1]$ range. As such, the total reward at any state is always equal to 1.0 according to (6) but distributed across all known classes at that state. We denote the thus far known classes at state s_n by $Y_{s_n} \subseteq Y$.

$$\sum_{y_j \in Y_{s_n}} p_j = 1.0 \quad (6)$$

B. Efficiency & Convergence of the Training Process

1) *Parallel Query Updates*: For any given state s_n , there exists a set of super-set states that lead to the same state, if the appropriate query is used [3]. An example of this is shown in Table I for the mushroom data set.

TABLE I: Parallel Query Updates, Mushroom Dataset

Feature	State 1	State 2	State 3	State 4	State 5
cap-color	$E = 0.57$	w	w	w	w
bruises	t	$E = 0.60$	t	t	t
odor	a	a	$E = 0.53$	a	a
cap-shape	f	f	f	$E = 0.48$	f
Reward	0.45	0.58	0.47	0.23	0.74

In Table I, State 1 will query *cap-color* because the expected reward is 0.57, which is higher than the report's reward of 0.45. The same holds for states 2-4, for which the report rewards are also smaller than feature query rewards. Hence, they will all transition to State 5, retrieving the report's reward of 0.74, and updating the query's reward appropriately. Therefore, these additional queries can also be updated at each state transition, requiring less training for convergence.

2) *Parallel Report Updates*: During the end condition of a training cycle, the report actions are updated against the known class label. It is also valid to state that all report actions of visited states during that training cycle could also be updated because they are all super-sets of the final state [3].

Looking again at Table I, State 5 is a guaranteed end-scenario because there are no more features. Hence, if this state is reached, the report action will be utilized and the classification label's reports will be updated. However, to get to this state, any of States 1-4 must have been visited first. Hence, the report's expected rewards at States 1-4 can also be updated, thereby again speeding up convergence.

In theory this seems appropriate, however there exists a convergence problem by introducing reward before the end condition. During training, the super states (States 1-4) of the end state (State 5) may be visited more often causing expected rewards to converge earlier. Hence, a bias is formed, and training is unable to discover new classification labels deeper

in the state space. This is illustrated in Fig. 1 and Fig. 2, where the results of regular reward propagation and propagation with parallel report updates, respectively are displayed.

In particular, Fig. 1 and Fig. 2 show four states at different time points with the expected rewards of the queries and reports. Each state contains a set of features s , various queries F_i and reports R_j . For better clarity, only one report's expected reward Q_R is shown per state. Fig. 1 shows the normal progression and convergence of the expected rewards (and probabilities). Fig. 2 shows the progression and convergence problem when parallel report updates are utilized. Table II describes the process details across time.

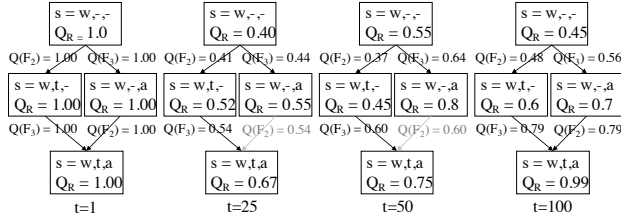


Fig. 1: Regular report updates: states over time

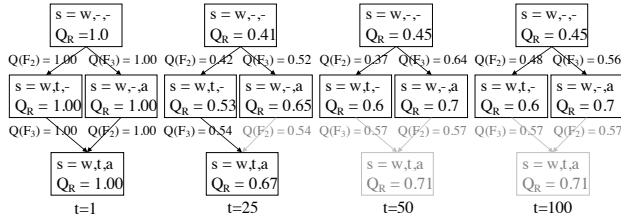


Fig. 2: Parallel report updates: states over time

TABLE II: Reward propagation in regular vs parallel report update processes - description of states over time

Time t	Regular Report Updates (Fig. 1)	Parallel Report Updates (Fig. 2)
1	Initial states created.	Initial states created.
25	A new classification label is encountered, sharing the reward between labels.	The expected rewards are converging faster. A new classification label is encountered, sharing the reward between labels.
50	Reward propagation continues from end state through the super states.	The super states have already converged, preventing exploration.
100	Convergence reached.	No changes occur.

C. Dealing with concept drifts

The Gini impurity index shown in (7) monitors the ongoing validity of the states, preventing the policy from guessing. Each time the reports are updated, the state's Gini impurity index is also updated according to (9) and normalized using the maximum possible Gini impurity index with (8), where n_R is the number of reports at that state. If this normalized value exceeds a user-defined threshold such as 0.99, the state has

a nearly even probability for each classification label, hence the reports and expected rewards are reset, allowing for local re-learning and handling of concept drifts.

$$GiniIndex = 1 - \sum_j p_{R_j}^2 \quad (7)$$

$$MaxGiniIndex = 1 - (1/n_R) \quad (8)$$

$$NormGiniIndex = GiniIndex / MaxGiniIndex \quad (9)$$

1) *Dimensionality*: A large concern of modeling the state as a combination of features of the instance, is the potentially large state space, and the heavy search requirements during state transitions. A few simple calculations can show that 10 features with 2 values result in $2^{10} = 10,24$ possible states. Likewise 3 value-options produces $3^{10} = 59,049$ possible states and 4 value-options produces $4^{10} = 1,048,576$ possible states. It is clear therefore that this could be problematic for problems with hundreds of features and a larger number of discrete values.

However, this is the exact reason why reinforcement learning is a good approach, as opposed to exhaustive techniques like a lookup table. As shown in the baseline example results of Fig. 3, the mushroom dataset did not need to explore all of these combinations. It learns quickly to avoid many state space ranges, and hence only explores a small subset. For reference, the mushroom dataset with 22 features and more than 4 discrete values per feature (5.2 average) has $1.219E+14$ state possibilities. However, training in this example only required 1465 states, which is $1.202E-9$ percent of the maximum possible state space and still provided 99.7% accuracy.

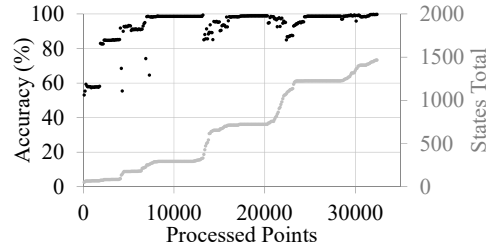


Fig. 3: ID 2 - Baseline, original file

It should be noted that other issues of high dimensionality still exist such as the potentially impractical search times. In a standard search, the state space may involve multiple filters to look up another state during transitions. Hence an alternative approach for locating states is necessary. To reduce search times of the state space, a hash code is generated from a state's feature-value pairs and then used as the key. In this way, a working state can be added to the state space easily and existing states can be located more quickly.

V. RL-BASED DECISION TREE CLASSIFICATION

There are two methods by which a new instance can be classified. The first, using the complete MDP policy according to Fig. 4 and Fig. 5. The second, summarizing the MDP policy

into a simplified decision tree according to Fig. 6 and then using standard decision tree deduction.

Each method has its advantages and disadvantages. Using the complete MDP policy allows for large flexibility because it compares all known features at each state, enabling it to handle incomplete instances. Naturally, a disadvantage is that this also requires more processing and may become slow if the feature space grows very large. Conversely, the summarized decision tree is fast but sacrifices handling of partial instances, because it contains only the most important features from the MDP and classifies within that specific feature space only.

```

procedure CLASSIFYBYMDP(dataVector)
  currState  $\leftarrow$  policy.rootState
  while (true) do
    bestQuery  $\leftarrow$  GetBestQuery(currState, dataVector)
    if (bestQuery  $\neq$  blank) then
      currState  $\leftarrow$  GetState(currState, bestQuery)
    else
      currLabels  $\leftarrow$  currState.Labels
      break loop
    end if
  end while
  theLabel  $\leftarrow$  PickLabel(currLabels)
  return theLabel
end procedure

```

Fig. 4: Classify an instance by MDP policy

```

procedure GETBESTQUERY(currState, dataVector)
  for all (feature  $\in$  dataVector) do
    validQuery  $\leftarrow$  currState.GetQuery(feature)
    valQueries.Insert(validQuery)
  end for
  for all (query  $\in$  valQueries) do
    qryReward  $\leftarrow$  query.Reward
    lblReward  $\leftarrow$  Labels.Rewards.Min
    if (qryReward  $<$  lblReward) then
      valQueries.Remove(query)
    end if
  end for
  if (valQueries.Count = 0) then return blank
  end if
  bestQuery  $\leftarrow$  valQueries.Max
  return bestQuery
end procedure

```

Fig. 5: Compare labels and queries to select best query

VI. EXPERIMENTS

The goal of the experiments is to explore the effects of training order, discount factor, exploration rate and feature space on the accuracy and created states, ideally providing optimal parameter values. Another important impact is how quickly the model can re-learn during concept drift. Other decision tree induction techniques are not compared, as accuracy and speed benefits are already demonstrated in [3]. In particular, the authors show that RLDT can achieve an accuracy of $97.74\% \pm 3.01$ using an average of just 1.1 queries. Additionally they compare RLDT to StARMiner Tree (ST),

```

procedure POLICYTOTREE(policy)
  rootState  $\leftarrow$  policy.rootState
  rootNode  $\leftarrow$  (new)TreeNode
  PolicyToTree(rootState, rootNode)
  return rootNode
end procedure
procedure POLICYTOTREE(currState, parentNode)
  bestGroupQueries = getQueriesMaxReward(currState)
  lblReward  $\leftarrow$  currState.Labels.Rewards.Min
  for all (query  $\in$  bestGroupQueries) do
    qryReward  $\leftarrow$  query.Reward
    if (qryReward  $<$  lblReward) then
      bestGroupQueries.Remove(query)
    end if
  end for
  featureNode  $\leftarrow$  (new)TreeNode
  parentNode.SubNodes.Insert(featureNode)
  for all (query  $\in$  bestGroupQueries) do
    queryNode  $\leftarrow$  (new)TreeNode
    parentNode.SubNodes.Insert(queryNode)
    currState  $\leftarrow$  getState(currState, query)
    PolicyToTree(currState, queryNode)
  end for
  for all (label  $\in$  currState.Labels) do
    featureNode.Leaves.Insert(label)
  end for
  return parentNode
end procedure

```

Fig. 6: Summarize the MDP policy to a decision tree

Automatic StARMiner Tree (AST), VFDT and VFDTcNB, where it has both higher final accuracy and mean accuracy.

A. Datasets

Three different data sets were considered for evaluation but only the Mushroom¹ dataset is used for the chosen evaluation topics because of its discrete values. Although RLDT can produce trees for datasets such as the Wine² and Attrition³ datasets with both discrete and continuous data types, the resulting trees are large and complex, essentially only memorizing the data. A solution would be to discretize the numeric fields, for example by equal-width or equal-depth histograms. However, since this was not the focus of the experiments, it is left for future work on how to effectively deal with continuous variables.

B. Experimental settings

Various streams of the original Mushroom dataset are simulated for the evaluation topics. A list of these different variations are shown in table IV. Since the original dataset has no temporal information, different orders by class are generated. Variations with increased feature space have additional features with 3-5 discrete values options randomly distributed across class labels.

As this evaluation involves a parameter study across multiple factors, a set of default parameters are defined. The

¹ Available at <https://archive.ics.uci.edu/ml/datasets/mushroom>

² Available at <https://archive.ics.uci.edu/ml/datasets/wine+quality>

³ Available at <https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/>

default testing values as well as the testing ranges are shown in table III. The default discount factor was set to $\gamma = 0.85$ to align with common discount factors in reinforcement learning [6]. The exploration rate was set to $\epsilon = 0.0$ because new queries are created optimistically. Parallel query updates are enabled to reduce testing time and parallel report updates are disabled because of the convergence problem discussed in Section IV-B2.

TABLE III: Testing Parameters and Ranges

Parameter	Default value	Evaluation Range
Exploration Rate:	0.00	0 to 0.9
Discount Factor:	0.85	0 to 0.99
Data Order:	Random	Asc, Dsc, Random (by class)
Parallel Query Updates:	Enabled	-
Parallel Report Updates:	Disabled	-

TABLE IV: Derived Mushroom Data Sets

Dataset Variation	Data Order	# Instances	# Features	Possible States
D1	Original	8125	22	1.219E+14
D6	Original	8125	7	45360
D7	Ascending, by class	8125	22	1.219E+14
D8	Descending, by class	8125	22	1.219E+14
D9	Random	8125	22	1.219E+14
D10	Random (classes flipped)	8125	22	1.219E+14
D11	Random	8125	145	1.10E+100
D12	Random	8125	132	1.04E+90
D13	Random	8125	117	1.03E+80
D14	Random	8125	103	1.03E+70
D15	Random	8125	88	1.08E+60
D16	Random	8125	74	1.06E+50
D17	Random	8125	59	1.06E+40
D18	Random	8125	45	1.05E+30
D19	Random	8125	31	1.09E+20

C. Experimental results

1) *Effect of training order:* Training is performed comparing the baseline results (Fig. 3) with data read in random, ascending, and descending order by class. In other words, all instances for one class are fed to the learner one by one. A summary of the results are shown in table V and Fig. 7.

The effect of training order by class significantly influences the time needed for learning. With ascending order (Fig. 7a), training requires many more processed instances to reach comparable accuracy to the baseline (Fig. 3). Interestingly, for descending order (Fig. 7b) the policy converges faster but does not achieve the 99% accuracy.

A randomly ordered stream (Fig. 7d) is trained much faster; high accuracy is achieved within the first half of only 1 pass. In Fig. 7c, a different kind of randomness is introduced through an $\epsilon = 10\%$ exploration rate. This causes training to converge slightly slower than the baseline (Fig. 3), but still achieves good performance.

TABLE V: Effect of training order on accuracy and total states

Dataset	Training Order	Exp. Rate (ϵ)	Total Passes	Total States	Accuracy [%]
D1	Baseline, original file	0.0	4	1465	99.7
D7	Ascending by class	0.0	40	4279	99.6
D8	Descending by class	0.0	6	889	98.5
D9	Random by class	0.0	1	787	99.4
D7	Ascending by class	0.1	7	7682	99.4

2) *Effect of discount factor:* The discount factor is adjusted between 0.1 and 0.95, to determine its effect on classification accuracy and generated states because it can influence the path tree lengths within the policy. Discount factors below 0.5 did not produce a trained policy, hence they are omitted. Only the first pass is considered, and parallel query updates are disabled to allow slower convergence and easier to interpret results.

Fig. 8 and Fig. 9 show the discount factor is best between 0.80 to 0.95, which is expected and fits to common reinforcement learning practices [6]. The region of 0.55 to 0.75 has a reversal point, indicating there may be a local optimality, hence this region produces inconsistent results and is not recommended. Finally, discount factors of 0.75 to 0.95 shift results toward higher accuracy and lower generated states.

3) *Effect of exploration rate:* The exploration rate is adjusted from 0 to 0.9, to determine its effect on the accuracy, required states, and required queries which provide insight for processing times and memory requirements. All processing is observed after one pass, and both parallel query updates and parallel report updates are disabled.

As shown in Fig. 12, even with no exploration rate, training can still effectively occur with just one pass producing a stable accuracy of 94.8%. However, with even just a 0.01 (1%) exploration rate, mistakes are introduced, and continued learning is necessary to correct explorations. However, this minor exploration rate enables greater accuracy.

As expected, utilizing an exploration rate requires more states and queries. Fig. 13 shows the number of states increase according to a 6th order polynomial. Fig. 14 shows the maximum, average, and minimum number of queries, which again increase very quickly above an exploration rate of 0.4.

4) *Effect of concept drift:* 2Two data sets are used, one with the original labels (D9) and one with reversed labels (D10), from table IV. The policy is initially trained with 1 pass of the original labels. Next, the two data sets are alternated back and forth to simulate full concept drift, 2 passes per turn. As shown in Fig. 15, the switch occurs 3 times and each time the policy quickly relearns the new DT with high accuracy above 99%. However, more states are still created indicating that prior knowledge is only being partly used.

5) *Effect of feature space:* As previously mentioned, there is significant concern about the effect of the instance's number of features on the processing time and state space requirements. Theoretically reinforcement learning should aid with this dimensionality problem because new paths are only explored when the accuracy is lower than desired. It is

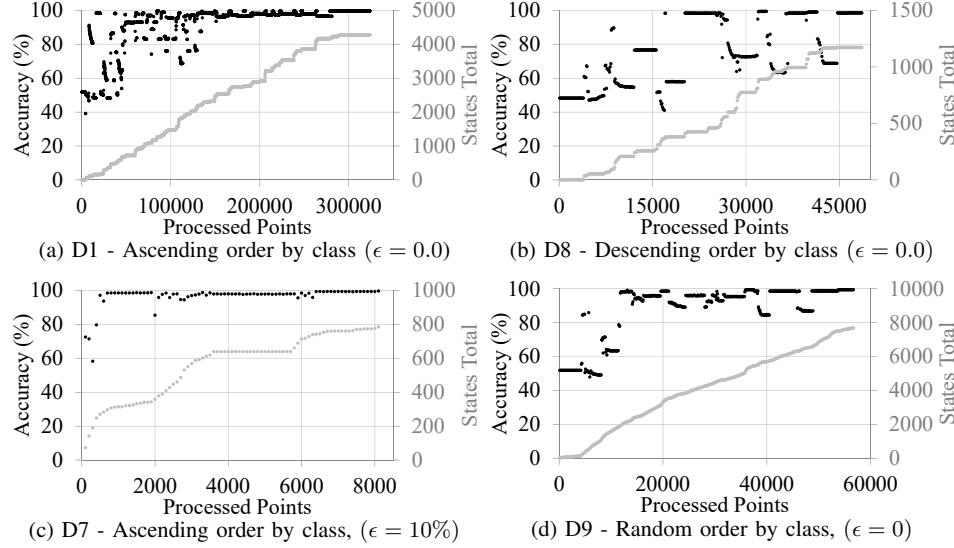


Fig. 7: Effect of training order on accuracy (black) and number of created states (gray), under different exploration rates ϵ

expected and additionally shown in Fig. 16 that a large number of unimportant features for classification do not drastically increase the state space.

This is tested with mushroom dataset variation which includes 20-120 additional features with random values with 3-5 unique values per feature. These additional features, since random, do not affect the resulting output of the tree and the results are still verifiable. As shown in Fig. 16, the potential state space is increased from 1 to $1E+100$. The processing time flattens off then later begins to increase. However, the number of created states continuously increases. Although the created states increases, table VI shows that the percentage of explored state space is always nearly zero.

TABLE VI: Processing Time and States vs Feature Space

Possible Feature Combinations	Processing Time [s]	Relative Time	States Created	Created Space [%]
$1.1E+100$	88.228	26.52	2200	$2.00E-95$
$1.04E+90$	80.658	24.24	1924	$1.85E-85$
$1.03E+80$	75.151	22.59	1724	$1.67E-75$
$1.03E+70$	48.811	14.67	1551	$1.51E-65$
$1.08E+60$	34.630	10.41	1490	$1.38E-55$
$1.06E+50$	23.942	7.20	1538	$1.45E-45$
$1.06E+40$	16.827	5.06	1574	$1.48E-35$
$1.05E+30$	9.348	2.81	1459	$1.39E-25$
$1.09E+20$	5.638	1.69	1235	$1.13E-15$
$1.22E+14$	3.327	1.00	787	$6.45E-10$

VII. CONCLUSIONS AND OUTLOOK

An RL-based decision tree induction method for data streams is proposed, which extends the work of [3] by allowing for local relearning within tree paths that become outdated over the course of the stream, due to concept drifts or

feature replacement in the underlying population. Preliminary results show that an RL-based decision tree can be used to successfully avoid the downsides of traditional decision tree methods, which require greedy separation techniques, as well as produces a compact and accurate decision tree.

Several improvements and extensions can be pursued as part of future work. First, improvements with respect to the time and space efficiency of the model are possible by removing, for example, non-recently used or low reward states.

For classification, it may also be possible to produce a partially simplified decision tree. Features could be flagged as more likely to become missing and be kept in the simplified decision tree, allowing classification in more situations with incomplete data while retaining faster processing speeds.

Finally, an integrated extension to continuous features is planned. Rather than discretizing the features at a preprocessing step, which would require apriori knowledge about the value ranges and would result in fixed bins, an integrated bin generation in the learning process is planned. This way, the value ranges will be created dynamically based on the incoming data and learning must not be restarted.

All code within this project is available to the community at <https://github.com/chriswblake/Reinforcement-Learning-Based-Decision-Tree> for reproducibility purposes and to further encourage the development of RL-based DT inductions methods for data streams.

ACKNOWLEDGEMENTS

The work was partially funded by the European Commission for the ERC Advanced Grant ALEXANDRIA under grant No. 339233 and inspired by the German Research Foundation (DFG) project OSCAR (Opinion Stream Classification with Ensembles and Active learners) for which the last author is Co-Principal Investigator.

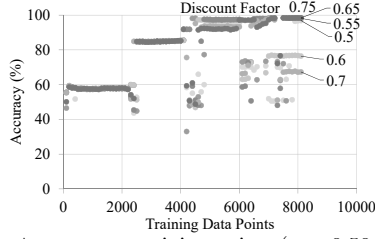


Fig. 8: Accuracy vs training points ($\gamma = 0.50 - 0.75$)

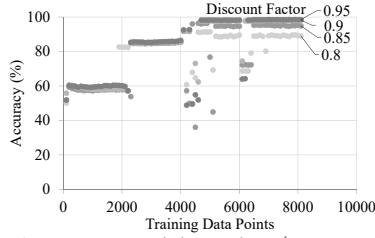


Fig. 9: Accuracy vs training points ($\gamma = 0.80 - 0.95$)

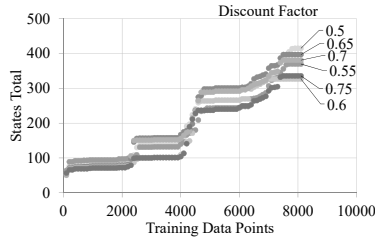


Fig. 10: Created states vs training points ($\gamma = 0.50 - 0.75$)

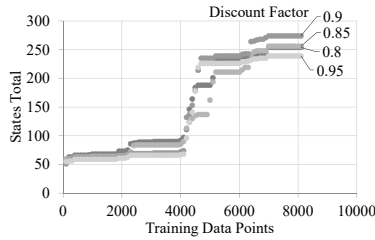


Fig. 11: Created states vs training points ($\gamma = 0.80 - 0.95$)

REFERENCES

- [1] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, and Ricard Gavaldà. Improving adaptive bagging methods for evolving data streams. In Zhi-Hua Zhou and Takashi Washio, editors, *Advances in Machine Learning*, pages 23–37, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
- [2] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.
- [3] Abhinav Garlapati, Aditi Raghunathan, Vaishnavh Nagarajan, and Balaraman Ravindran. A Reinforcement Learning Approach to Online Learning of Decision Trees. Technical report, Department of Computer Science, Indian Institute of Technology, Madras, 2015.
- [4] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, pages 97–106, New York, NY, USA, 2001. ACM.
- [5] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.
- [6] Richard S Sutton and Andrew G Barto. *Reinforcement Learning : An Introduction*. The MIT Press, Cambridge, Massachusetts, draft edition.
- [7] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998.

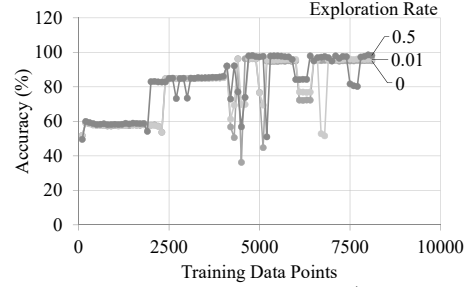


Fig. 12: Accuracy vs training points ($\epsilon = 0.0 - 0.5$)

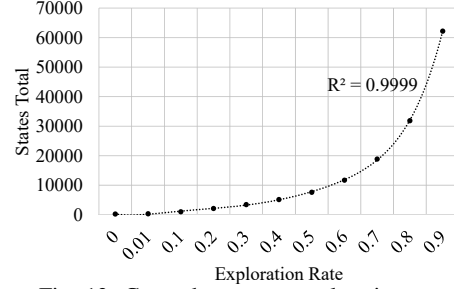


Fig. 13: Created states vs exploration rate

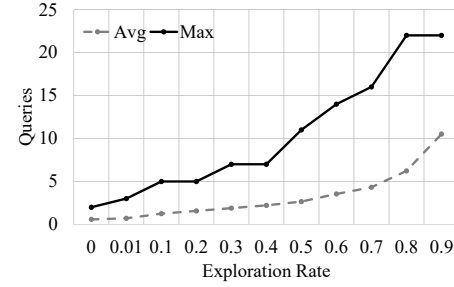


Fig. 14: Queries vs exploration rate

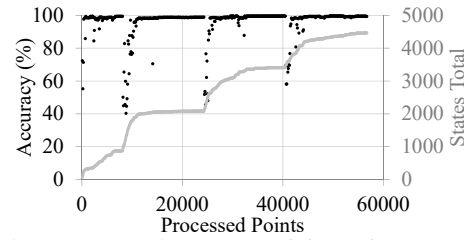


Fig. 15: Accuracy and states vs training points, concept drift

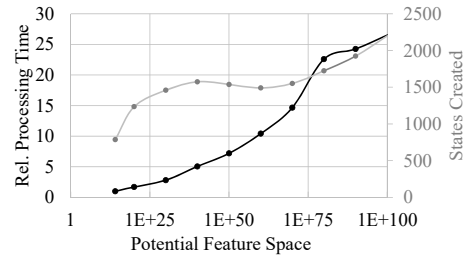


Fig. 16: Processing time and created states vs feature space