Online Machine Learning Techniques for Coq: A Comparison*

Liao Zhang[1,3], Lasse Blaauwbroek[1,2], Bartosz Piotrowski[1,4], Prokop Černý[1], Cezary Kaliszyk[3,4] and Josef Urban[1]

[1] Czech Technical University, Prague, Czech Republic
[2] Radboud University, Nijmegen, The Netherlands
[3] University of Innsbruck, Austria
[4] University of Warsaw, Poland

Abstract. We present a comparison of several online machine learning techniques for tactical learning and proving in the Coq proof assistant. This work builds on top of Tactician, a plugin for Coq that learns from proofs written by the user to synthesize new proofs. Learning happens in an online manner, meaning that Tactician's machine learning model is updated immediately every time the user performs a step in an interactive proof. This has important advantages compared to the more studied offline learning systems: (1) it provides the user with a seamless, interactive experience with Tactician and, (2) it takes advantage of locality of proof similarity, which means that proofs similar to the current proof are likely to be found close by. We implement two online methods, namely approximate k-nearest neighbors based on locality sensitive hashing forests and random decision forests. Additionally, we conduct experiments with gradient boosted trees in an offline setting using XGBoost. We compare the relative performance of Tactician using these three learning methods on Coq's standard library.

Keywords: Interactive Theorem Proving · Coq · Machine Learning · Online Learning · Gradient Boosted Trees · Random Forest

1 Introduction

The users of interactive theorem proving systems are in dire need of a digital sidekick, which helps them reduce the time spent proving the mundane parts of their theories, cutting down on the man-hours needed to turn an informal theory into a formal one. The obvious way of creating such a digital assistant is using machine learning. However, creating a practically usable assistant comes with some requirements that are not necessarily conducive to the most trendy machine learning techniques, such as deep learning.

* This work was supported by the ERC grant no. 714034 *SMART*, by the European Regional Development Fund under the project AI&Reasoning (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000466), and by the Ministry of Education, Youth and Sports within the dedicated program ERC CZ under the project POSTMAN no. LL1902.

The environment provided by ITPs is highly dynamic, as it maintains an ever-changing global context of definitions, lemmas, and custom tactics. Hence, proving lemmas within such environments requires intimate knowledge of all the defined objects within the global context. This is contrasted by—for example—the game of chess; even though the search space is enormous, the pieces always move according to the same rules, and no new kinds of pieces can be added. Additionally, the interactive nature of ITPs demands that machine learning techniques do not need absurd amounts of time and resources to train (unless a pre-trained model is highly generic and widely applicable across domains; something that has not been achieved yet). In this paper, we are interested in online learning techniques that quickly learn from user input and immediately utilize this information. We do this in the context of the Coq proof assistant [26] and specifically Tactician [5]—a plugin for Coq that is designed to learn from the proofs written by a user and apply that knowledge to prove new lemmas.

Tactician performs a number of functions, such as proof recording, tactic prediction, proof search, and proof reconstruction. In this paper, we focus on tactic prediction. For this, we need a machine learning technique that accepts as input a database of proofs, represented as pairs containing a proof state and the tactic that was used to advance the proof. From this database, a machine learning model is built. The machine learning task is to predict an appropriate tactic when given a proof state. Because the model needs to operate in an interactive environment, we pose four requirements the learning technique needs to satisfy:

1. The model (datastructure) needs to support dynamic updates. That is, the addition of a new pair of a proof state and tactic to the current model needs to be done in (near) constant time.
2. The model should limit its memory usage to fit in a consumer laptop. We have used the arbitrary limit of 4 GB.
3. The model should support querying in (near) constant time.
4. The model should be persistent (in the functional programming sense [11]). This enables the model to be synchronized with the interactive Coq document, in which the user can navigate back and forth.

1.1 Contributions

In this work, we have implemented two online learning models. An improved version of the locality sensitive hashing scheme for k-nearest neighbors is described in detail in Section 3.1. An implementation of random forest is described in Section 3.2. In Section 4, we evaluate both models, comparing the number of lemmas of Coq's standard library they can prove in a chronological setting (i.e., emulating the growing library).

In addition to the online models, as a proof of concept, we also experiment in an offline fashion with boosted trees, specifically XGBoost [8] in Section 3.3. Even though the model learned by XGBoost cannot be used directly in the online setting described above, boosted trees are today among the strongest learning methods. Online algorithms for boosted trees do exist [27], and we intend to implement them in the future.

The techniques described here require representing proof states as feature vectors. Tactician already supported proof state representation using simple hand-rolled features [4]. In addition, Section 2 describes our addition of more advanced features of the proof states, which are shown to improve Tactician's performance in Section 4.

2 Tactic and Proof State Representation

To build a learning model, we need to characterize proof states and the tactics applied to them. To represent tactics, we first perform basic decompositions and simplifications and denote the resulting atomic tactics by their hashes [4].

Tactician's original proof state features [4] consist merely of identifiers and adjacent identifier pairs in the abstract syntax tree (AST). Various other, more advanced features have been considered for automated reasoning systems built over large formal mathematical knowledge bases [9,14,20]. To enhance the performance of Tactician, we modify the old feature set and define new features as follows.

Top-down Oriented AST Walks We add top-down oriented walks in the AST of length up to 3 with syntax placeholders. For instance, the unit clause $f(g(x))$ will contain the features:

```
f:AppFun, g:AppFun, x:AppArg, f:AppFun(g:AppFun),
g:AppFun(x:AppArg), f:AppFun(g:AppFun(x:AppArg))
```

The feature `g:AppFun` indicates that g is able to act as a function in the term tree, and `x:AppArg` means that x is only an argument of a function.

Vertical Abstracted Walks We add vertical walks in the term tree from the root to atoms in which nonatomic nodes are substituted by their syntax roles. For the term $f_1(f_2(f_3(a)))$, we can convert each function symbol to `AppFun` whereas the atom a is transformed to `a:AppArg` as above. Subsequently, we can export this as the feature `AppFun(AppFun(AppFun(a:AppArg)))`. Such abstracted features are designed to better capture the overall abstract structure of the AST.

Top-level Structures We add top-level patterns by replacing the atomic nodes and substructures deeper than level 2 with a single symbol `X`. Additionally, to separate the function body and arguments, we append the arity of the function to the corresponding converted symbol. As an example, consider the term $f(g(b,c),a)$ consisting of atoms a, b, c, f, g. We first replace a, f, g with `X` because they are atomic. We further transform f and g to `X2` according to the number of their arguments. However, b and c break the depth constraint and should be merged to a single `X`. Finally, the concrete term is converted to an abstract structure `X2(X2(X),X)`. Abstracting a term to its top-level structure is useful for determining whether a "logical" tactic should be applied. As an illustration, the presence of $X \wedge X$ in the goal often indicates that we should perform case analysis by the `split` tactic. Since we typically do not need all the nodes of a term to decide such structural information, and we want to balance the generalization with specificity, we use the maximum depth 2.

Premise and Goal Separation Because local hypotheses typically play a very different role than the conclusion of a proof state, we separate their feature spaces. This can be done by serially numbering the features and adding a sufficiently large constant to the goal features.

Adding Occurrence Counts In the first version of Tactician, we have used only a simple boolean version of the features. We try to improve on this by adding the number of occurrences of each feature in the proof state.

3 Prediction Models

3.1 Locality Sensitive Hashing Forests for Online *k*-NN

One of the simplest methods to find correlations between proof states is to define a metric or similarity function $d(x, y)$ on the proof states. One can then extract an ordered list of length k from a database of proof states that are as similar as possible to the reference proof state according to d. Assuming that d does a good job identifying similar proof states, one can then use tactics known to be useful in a known proof state for an unseen proof state. In this paper, we refer to this technique as the k-nearest neighbor (k-NN) method (even though this terminology is somewhat overloaded in the literature).

Our distance function is based on the features described in Section 2. We compare these features using the Jaccard index $\mathrm{J}(f_1, f_2)$. Optionally, features can be weighted using the TfIdf statistic [18], in which case the generalized index $\mathrm{J}_w(f_1, f_2)$ is used.

$$\mathrm{J}(f_1, f_2) = \frac{|f_1 \cap f_2|}{|f_1 \cup f_2|} \quad \mathrm{tfidf}(x) = \log \frac{N}{|x|_N} \quad \mathrm{J}_w(f_1, f_2) = \frac{\sum_{x \in f_1 \cap f_2} \mathrm{tfidf}(x)}{\sum_{x \in f_1 \cup f_2} \mathrm{tfidf}(x)}$$

Here N is the database size, and $|x|_N$ is the number of times feature x occurs in the database. In previous work, we have made a more detailed comparison of similarity functions [4].

A naive implementation of the k-NN method is not very useful in the online setting because the time complexity for a query grows linearly with the size of the database. Indexing methods, such as k-d trees, exist to speed up queries [3]. However, these methods do not scale well when the dimensionality of the data increases [17]. In this work, we instead implement an approximate version of the k-NN method based on Locality Sensitive Hashing (LSH) [16]. This is an upgrade of our previous LSH implementation that was not persistent and was slower. We also describe our functional implementation of the method in detail for the first time here.

The essential idea of this technique is to hash feature vectors into buckets using a family of hash functions that guarantee that similar vectors hash to the same bucket with high probability (according to the given similarity function). To find a k-NN approximation, one can simply return the contents of the bucket corresponding to the current proof state. For the Jaccard index, the appropriate family of hash functions are the MinHash functions [7].

The downside of the naive LSH method is that its parameters are difficult to tune. The probability that the vectors that hash to the same bucket are similar can be increased by associating more than one hash function to the bucket. All values of the hash functions then need to pair-wise agree for the items in the bucket. However, this will naturally decrease the size of the bucket, lowering the number of examples k (of k-NN) that can be retrieved. The parameter k can be increased again by simply maintaining multiple independent bucketing datastructures. Tuning these parameters is critically dependent on the size of the database, the length of the feature vectors, and the desired value of k. To overcome this, we implement a highly efficient, persistent, functional variant of Locality Sensitive Hashing Forest [2] (LSHF), which is able to tune these parameters automatically, leaving (almost) no parameters to be tuned manually. Below we give a high-level overview of the algorithm as it is modified for a functional setting. For a more in-depth discussion on the correctness of the algorithm, we refer to the previous reference.

LSHFs consist of a forest (collection) of tries $\mathcal{T}_1 \dots \mathcal{T}_n$. Every trie has an associated hash function h_i that is a member of a (near) universal hashing family mapping a feature down to a single bit (a hash function mapping to an integer can be used by taking the result modulus two). To add a new example to this model, it is inserted into each trie according to a path (sequence) of bits. Every bit of this path can be shown to be locally sensitive for the Jaccard index [2]. The path of an example is calculated using the set of features that represents the proof state in the example.

$$\text{path}_i(f) = \text{sort}(\{h_i(x) \mid x \in f\})$$

For a given trie $\mathcal{T}$, the subtrie starting at a given path $b_1 \dots b_m$ can be seen as the bucket to which examples that agree on the hashes $b_1 \dots b_m$ are assigned. Longer paths point to smaller buckets containing less similar examples, while shorter paths point to larger buckets containing increasingly similar examples. Hence, to retrieve the neighbors of a proof state with features f, one should start by finding examples that share the entire path of f. To retrieve more examples, one starts collecting the subtrees starting at smaller and smaller prefixes of $\text{path}_i(f)$. To increase the accuracy and number of examples retrieved, this procedure can be performed on multiple tries simultaneously, as outlined in Algorithm 1.

Tuning the LSHF model consists mainly of choosing the appropriate number of tries that maximizes the speed versus accuracy trade-off. Experiments show that 11 trees is the optimal value. Additionally, for efficiency reasons, it is a good idea to set a limit on the depth of the tries to prevent highly similar examples from creating a deep trie. For our dataset, a maximum depth of 20 is sufficient.

3.2 Online Random Forest

Random forests are a popular machine learning method combining many randomized decision trees into one ensemble, which produces predictions via voting [6]. Even though the decision trees are not strong learners on their own, because

Algorithm 1 Querying the Locality Sensitive Hashing Forest

```
1: function QueryLSHF(F, f)                          ▷ F a forest, f a feature set
2:     P ← ⟨path_i(f) : i ∈ [1..|F|]⟩
3:     neighbors ← FilterDuplicates(SimultaneousDescend(F, P))
4:     Optionally re-sort neighbors according to real Jaccard index
5: function SimultaneousDescend(F, P)
6:     F_rel ← ⟨ if head(P) then left(T) else right(T) : T ∈ F when not leaf(T) ⟩
7:     F_irrel ← ⟨ if leaf(T) then T elseif head(P) then right(T) else left(T) : T ∈ F ⟩
8:     if F_rel is empty then
9:         neighbors ← empty list
10:    else
11:        P' ← ⟨tail(P_i) : i ∈ [1..n]⟩
12:        neighbors ← SimultaneousDescend(F_rel, P')
13:    if |neighbors| ≥ k then
14:        return neighbors
15:    else
16:        return Append(neighbors, Concatenate(⟨ Collect(T : T ∈ F_irrel⟩)))
```

they are intentionally decorrelated, the voting procedure greatly improves on top of their individual predictive performance. The decision trees consist of internal nodes labeled by decision rules and leaves labeled by examples. In our case, these are tactics to be applied in the proofs.

Random forests are a versatile method that requires little tuning of its hyperparameters. Their architecture is also relatively simple, which makes it easy to provide a custom OCaml implementation easily integrable with Tactician, adhering to its requirement of avoiding mutable data structures. Direct usage of existing random forest implementations is impossible due to challenges in Tactician's learning setting. These challenges are: (1) numerous sparse features, (2) the necessity of online learning, as detailed in the next two paragraphs.

The decision rules in nodes of the decision trees are based on the features of the training examples. These rules are chosen to maximize the *information gain*, i.e., to minimize the *impurity* of the set of labels in the node.[5] There are more than $37,000$ binary and sparse features in Tactician. Since the learner integrated with Tactician needs to be fast, one needs to be careful when optimizing the splits in the tree nodes.

Random forests are typically trained in an offline manner where the whole training data is available at the beginning of the training. In Tactician this would be quite suboptimal. To take advantage of the locality of proof similarity and to be able to use data coming from new proofs written by a user, we want to immediately update the machine learning model after each proof.

There are approaches to turn random forests into online learners [10,25] which inspired our implementation. The authors of [10] propose a methodology where new training examples are passed to the leaves of the decision trees, and under

[5] If we have labels $\{a, a, b, b, b\}$, ideally, we would like to produce a split which passes all the examples with label a to one side and the examples with b to the other side.

Algorithm 2 Adding training a example e to a decision tree $\mathcal{T}$

```
1: function AddExampleToTree(𝒯, e)
2:     match 𝒯 with
3:         Node(ℛ, 𝒯_l, 𝒯_r):          ▷ ℛ – binary rule, 𝒯_l, 𝒯_r – left and right subtrees
4:             match ℛ(e) with
5:                 Left: return Node(ℛ, AddExampleToTree(𝒯_l, e), 𝒯_r)
6:                 Right: return Node(ℛ, 𝒯_l, AddExampleToTree(𝒯_r, e))
7:         Leaf(l, ℰ):                  ▷ l – label/tactic, ℰ – examples
8:             ℰ ← Append(ℰ, e)
9:             if SplitCondition(ℰ) then
10:                ℛ ← GenerateSplitRule(ℰ)
11:                ℰ_l, ℰ_r ← Split(ℛ, ℰ)
12:                l_l ← label of random example from ℰ_l
13:                l_r ← label of random example from ℰ_r
14:                return Node(ℛ, Leaf(l_l, ℰ_l), Leaf(l_r, ℰ_r))
15:            else
16:                return Leaf(l, ℰ)
```

certain statistical conditions, the leaf is split and converted to a new decision node followed by two new leaves. We take a similar approach, but deciding a split in our implementation is simpler and computationally cheaper.

The pseudocode describing our implementation is presented below. Algorithm 2 shows how the training examples are added to the decision trees. A new training example is passed down the tree to one of its leaves. The trajectory of this pass is governed by binary decision rules in the nodes of the tree. Each rule checks whether a given feature is present in the example. Once the example reaches a leaf, it is saved there, and a decision is made whether to extend the tree (using function SplitCondition). This happens only when the Gini Impurity measure [21] on the set of examples in the leaves is greater than a given impurity threshold i (a hyperparameter of the model). When the split is done, the leaf becomes an internal node with a new split rule, and the collected examples from the leaf are passed down to the two new leaves. The new rule (an output from GenerateSplitRule) is produced in the following way. N features are selected from the features of the examples, where N is the square root of the number of examples. The selection of the features is randomized and made in such a way that features that are distinguishing between the examples have higher probability: First, we randomly select two examples from the leaf, and then we randomly select a feature from the difference of sets of features of the two examples. Among such selected features, the one maximizing the *information gain* [21] of the split rule based on it is selected. The two new leaves get labels randomly selected from the examples belonging to the given leaf.

When adding an example to a random forest (Algorithm 3), first, a decision is made whether a new tree (in the form of a single leaf) should be added to the forest. It happens with probability $\frac{1}{n}$, where n is the number of trees in the forest under the condition that n is lower than a given threshold.

Algorithm 3 Adding a training example e to a random forest $\mathcal{F}$

```
1: function AddExampleToForest(F, e, n_max)        ▷ n_max – max number of trees
2:     n ← number of trees in F
3:     m ← random number from {1, ... n}
4:     F_updated ← empty list
5:     if n < n_max and m = 1 then
6:         T ← leaf labeled by tactic used in e
7:         F_updated ← Append(F_updated, T)
8:     for all T ∈ F do
9:         T ← AddExampleToTree(T, e)
10:        F_updated ← Append(F_updated, T)
11:    return F_updated
```

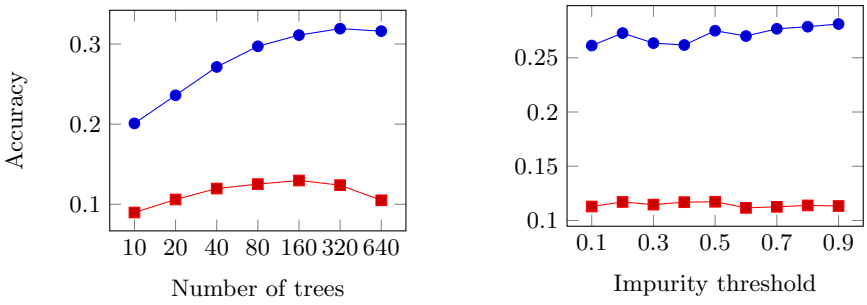
Algorithm 4 Predicting labels for unlabeled e in the random forest $\mathcal{F}$

```
1: function PredictForest(F, e)
2:     P ← empty list                              ▷ P – predictions
3:     for all T ∈ F do
4:         t ← PredictTree(e)
5:         append t to P
6:     R ← Vote(P)                                 ▷ R – ranking of tactics
7:     return R
8: function PredictTree(T, e)
9:     match T with
10:        Node(R, T_l, T_r):
11:            match R(e) with
12:                Left: return PredictTree(T_l, e)
13:                Right: return PredictTree(T_r, e)
14:        Leaf(l, E): return l
```

Predicting a tactic for a given example with a random forest (Algorithm 4) is done in two steps. First, the example is passed to the leaves of all the trees and the labels (tactics) in the leaves are saved. Then the ranking of the tactics is made based on their frequencies.

Tuning Hyperparameters There are two hyperparameters in our implementation of random forests: (1) the maximal number of trees in the forest and (2) the impurity threshold for performing the node splits. To determine the influence of these parameters on the predictive power, we perform a grid search. For this, we randomly split the data that is not held out for testing (see Section 4.1) into a training and validation part. The results of the grid search are shown in Figure 1. The best numbers of trees are 160 (for top-1 accuracy) and 320 (for top-10 accuracy). We used these two values for the rest of the experiments. For the impurity threshold, it is difficult to see a visible trend in performance; thus we selected 0.5 as our default.

Fig. 1. Results of hyperparameter tuning for random forests. The blue circle corresponds to top-10 accuracy (how often the correct tactic was present in the first 10 predictions) whereas the red square corresponds to top-1 accuracy.



3.3 Boosted Trees

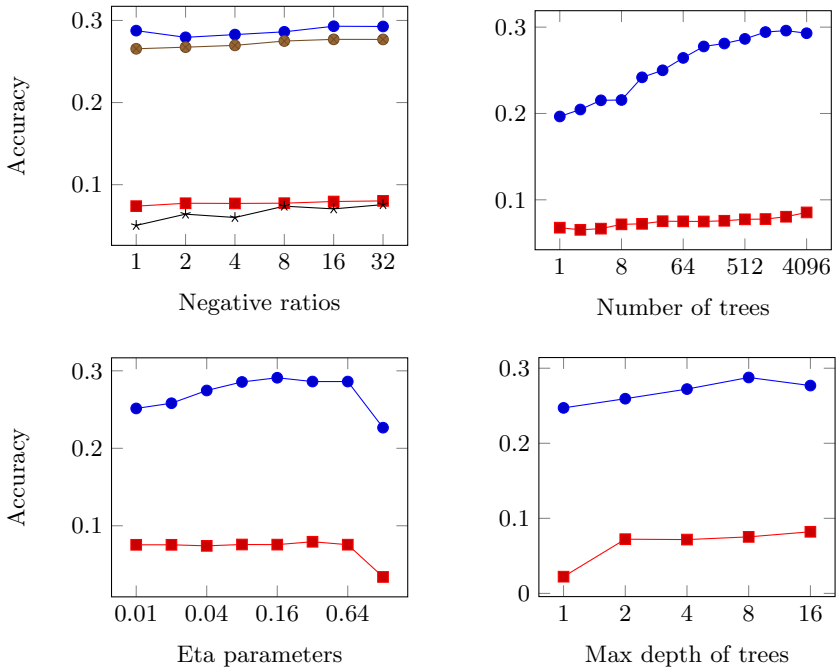
Gradient boosted decision trees are a state-of-the-art machine learning algorithm that transforms weak base learners, decision trees, into a method with stronger predictive power by appropriate combinations of the base models. One efficient and powerful implementation is the XGBoost library. Here, we perform some initial experiments in an offline setting for tactic prediction. Although XGBoost can at the moment not be directly integrated with Tactician, this gives us a useful baseline based on existing state-of-the-art technology. Below, we illustrate a procedure of developing our XGBoost model based on binary logistic regression.

The input to XGBoost is a sparse matrix containing rows with the format of (ϕ_P, ϕ_T) where ϕ_P includes the features of a proof state, and ϕ_T characterizes a tactic related to the proof state. We transform each proof state to a sparse feature vector ϕ_P containing the features' occurrence counts. Since there may be a large number of features in a given Coq development environment, which may hinder the efficiency of training and prediction, it is reasonable to decrease the dimension of the vectors. We hash the features to $20,000$ buckets by using the modulo of the feature's index. As above, we also remap the tactic hashes to a $20,000$-dimensional space separated from the state features.

The training examples get labels 1 or 0 based on the tactics being useful or not for the proof state. A tactic for a certain proof state is labeled as positive if it is exactly the one applied to this state in the library. In contrast, negative tactics are elements in the tactic space that differ from the positive instance. We obtain negative data by two approaches: *strong* negatives and *random* negatives. Strong negative instances are obtained by arbitrarily selecting a subset from the best-100 k-NN predictions for this state. In the other approach, negative instances are arbitrarily chosen from the entire tactic space.

With a trained gradient boosted trees model, we can predict the scores of the tactics for an unseen proof state P. First, the top-100 k-NN predictions are preselected. Then, for each tactic, we input (ϕ_P, ϕ_T) to the model to obtain the score of T. The tactics are then sorted according to their scores.

Fig. 2. Results of hyperparameter tuning for gradient boosted trees. In consistence with Figure 1, the blue circle (red square) corresponds to top-10 (top-1) accuracy, respectively. The graph of negative ratios contains two additional curves of random negative examples. The brown circle relates to top-10 accuracy, whereas the black star presents the results of top-1 accuracy.



Tuning Hyperparameters Similarly as for the random forest model (Section 3.2), we optimize the most important hyperparameters of the XGBoost training algorithm on the data coming from the non-sink nodes in the dependency graph of Coq's standard library (see Section 4.1). One essential parameter is the *ratio* of negative examples. Ratio n indicates that we generate n negative instances for each recorded proof state. Other influential parameters that we tune are: *eta* (learning-rate), *number of trees*, and *max depth*. Due to the limitations of computing resources, we assume a set of default parameters: $ratio = 8$, $eta = 0.2$, $number\ of\ trees = 500$, $max\ depth = 10$, and then separately modify each of these parameters to observe the influence caused by the change, which is depicted in Figure 2. Both strong and random negatives are evaluated. Obviously, strong negatives perform better than random negatives, and increasing the negative ratios will certainly lead to higher success rates. The figure also shows that a higher number of trees results in better performance. Learning rates are between 0.08 and 0.64 give good results. It is also apparent that deeper trees (at least 8) increase the accuracy.

Table 1. Performance of the three tested machine learning models in two types of evaluation: using a split of the dataset and a chronological evaluation through the dataset. top-n refers to the frequency of the correct tactic being present in the first n predictions from a machine learning model.

	Machine learning system					
	k-NN		Random Forest		XGBoost	
Evaluation type	top-1	top-10	top-1	top-10	top-1	top-10
split	18.8%	34.2%	32.1%	41.2%	18.2%	38.2%
chronological	17.3%	43.7%	29.9%	58.9%	18.2%	43.4%

Experimental Setup The XGBoost model is evaluated on the task of tactic prediction both in the split setting and the chronological setting (illustrated in Section 4). We use the strong negative examples and determine the final parameters—*ratio* = 16, *eta* = 0.2, *number of trees* = 1024, *max depth* = 10—for generating a model from non-sink nodes and use that to predict for sink nodes.

Since the entire dataset contains approximately 250,000 proof states, and it is time-consuming to generate a unique XGBoost model for each test case, we propose several ways to speed up the chronological evaluation. Instead of training on the data from all preceding states, we merely provide 1,000 instances occurring previously as the training data. According to the results of parameter tuning depicted in Figure 2, we decide on the hyperparameters—*ratio* = 4, *eta* = 0.2, *number of trees* = 256, *max depth* = 10—to balance the accuracy and efficiency.

4 Experimental Evaluation

To compare the performance of the described machine learning models, we perform three kinds of experiments: *split* evaluation, *chronological* evaluation, and evaluation in Tactician. Achieving good performance in the last type of evaluation is the main goal. All three machine learning models are evaluated in the first two kinds of experiments, while in Tactician we only evaluate k-NN and online random forest. This is because the XGBoost system, while being potentially the strongest machine learner among tested, may not be easily turned into an online learner and integrated into Tactician. We adopt the original features—term and term pairs—for evaluation outside Tactician, whereas both the original features and the new are tested on Tactician's benchmark. To determine the relative importance of the feature classes described in Section 2, we benchmark the addition of each class separately in Tactician. All evaluations are performed on data extracted from the standard library of Coq 8.11.

4.1 Split Evaluation

In the directed acyclic graph of dependencies of the Coq modules, there are 545 nodes. 104 of them are *sink nodes*, i.e., these are the modules that do not appear

among dependencies of any other module. We used these modules as final testing data for evaluation outside Tactician. The rest of the data was randomly split into training and validation parts and was used for parameter tuning of random forest and gradient boosted trees. The models with tuned hyperparameters were evaluated on the testing data. The results of the evaluation of the three tested models are shown in the first row of Table 1.

4.2 Chronological Evaluation

Although the split evaluation from the previous experiment is interesting, it does not correspond entirely to the Tactician's internal mode of operation. To simulate the real-world scenario in an offline setting, we create an individual model for each proof state by learning from all the previous states—data from dependent files and preceding lines in the local file. The second row of Table 1 presents the results of the evaluation in chronological order.

4.3 Evaluation in Tactician

Table 2 shows the results of the evaluation of two online learners—the k-NN and the random forest—within Tactician. The hyperparameters of the random forest model were chosen based on the grid search in Section 3.2. We run the proof search for every lemma in the library with a 40-second time limit on both the original and the improved features.

The random forest performed marginally better than k-NN on both kinds of features. With old features the k-NN proved 3831 lemmas (being 33.7% out of all 11370), whereas the random forest proved 4011 lemmas (35.3% of all). With the new features, both models performed better, and again, the random forest proved more lemmas (4117, 36.2% of all) than k-NN (3945, 34.7% of all).

It is somewhat surprising that the random forest, which performed much better than k-NN on the split in the offline evaluation, is only better by a small margin in Tactician. This may be related to the time and memory consumption of random forest, which may be higher than for k-NN on certain kinds of data.[6]

It is worth noting that k-NN and random forest resulted in quite different sets of proofs. The columns marked as *union* show that the size of the union of proofs constructed by the two models is significantly larger than the number of proofs found by each model separately. In total, both models resulted in 4503 (39.6%) proofs using old features and 4597 (40.4%) proofs using the new features.

4.4 Feature Evaluation

Table 3 depicts the influence of adding the new classes of features described in Section 2 to the previous baseline.[7] All of the newly produced features improve

[6] Doing the splits in the leaves has quadratic time complexity with respect to the number of examples stored in the leaf; sometimes it happens, that leaves of the trees store large number of examples.

[7] The results here are not directly comparable to those in Table 2 mainly due to the usage of a non-indexed version of k-NN in contrast to the algorithm presented in 1.

Table 2. Proving performance of two online learners integrated with Tactician, k-NN and random forest, in the Coq Standard Library. The percentages in the table correspond to the fraction of lemmas proved in a given Coq module. The columns *union* show what fraction of the lemmas was proved by at least one of the learners. RF is an abbreviation of random forest.

Coq module	**#Lemmas**	**Features type**					
		Original			New		
		k-NN	RF	*union*	k-NN	RF	*union*
All	**1137**	**33.7%**	**35.3%**	**39.6%**	**34.7%**	**36.2%**	**40.4%**
Arith	293	52%	59%	65%	56%	59%	66%
Bool	130	93%	87%	93%	92%	88%	92%
Classes	191	80%	76%	81%	79%	79%	83%
FSets	1137	32%	34%	37%	32%	35%	39%
Floats	5	20%	20%	20%	40%	19%	40%
Init	164	73%	51%	73%	73%	56%	73%
Lists	388	38%	43%	47%	38%	44%	49%
Logic	341	31%	27%	34%	32%	31%	35%
MSets	830	38%	40%	43%	36%	40%	43%
NArith	288	37%	43%	44%	35%	42%	47%
Numbers	2198	23%	22%	27%	24%	23%	27%
PArith	280	31%	40%	44%	35%	39%	45%
Program	28	75%	64%	75%	78%	66%	78%
QArith	295	33%	40%	43%	31%	39%	45%
Reals	1756	19%	23%	25%	21%	24%	26%
Relations	37	29%	24%	40%	27%	26%	29%
Setoids	4	1.00	1.00	1.00	1.00	97%	1.00
Sets	222	43%	42%	49%	49%	47%	53%
Sorting	136	26%	29%	33%	25%	30%	33%
Strings	74	22%	22%	27%	17%	14%	20%
Structures	390	45%	49%	54%	51%	51%	56%
Vectors	37	37%	29%	40%	21%	23%	27%
Wellfounded	36	19%	05%	19%	16%	13%	16%
ZArith	953	41%	46%	49%	40%	43%	46%
btauto	44	11%	20%	20%	20%	17%	22%
funind	4	75%	50%	75%	50%	73%	75%
micromega	339	21%	27%	29%	27%	25%	30%
nsatz	27	33%	33%	37%	40%	26%	40%
omega	37	40%	67%	67%	48%	63%	64%
rtauto	33	30%	39%	48%	33%	44%	51%
setoid_ring	362	21%	23%	26%	27%	27%	30%
ssr	311	68%	55%	69%	70%	57%	71%

the success rates. However, the top-down oriented AST walks contribute little, probably due to Tactician having already included term tree walks up to length 2. Every other modification obtains a reasonable improvement, which confirms the intuitions described in Section 2.

Table 3. Proving performance of each feature modification. $\mathcal{O}, \mathcal{W}, \mathcal{V}, \mathcal{T}, \mathcal{S}, \mathcal{C}$ denote original features, top-down oriented AST walks, vertical abstract walks, top-level structures, premise and goal separation, and adding feature occurrence, respectively. The symbol $\oplus$ denotes that we combine the original features and a new modification during the experiments.

Features	$\mathcal{O}$	$\mathcal{O} \oplus \mathcal{W}$	$\mathcal{O} \oplus \mathcal{V}$	$\mathcal{O} \oplus \mathcal{T}$	$\mathcal{O} \oplus \mathcal{S}$	$\mathcal{O} \oplus \mathcal{C}$
Success rates (%)	32.75	32.82	34.16	33.65	34.42	34.97

5 Related Work

Random forests were first used in the context of theorem proving by Färber [12], where multi-path querying of a random forest would improve on k-NN results for premise selection. Nagashima and He [22] proposed a proof method recommendation system for Isabelle/HOL based on decision trees on top of precisely engineered features. A small number of trees and features allowed for explainable recommendations. Frameworks based on random boosted trees (XGBoost, LightGBM) have also been used in automated reasoning, in the context of guiding tableaux connection proof search [19] and the superposition calculus proof search [9], as well as for handling negative examples in premise selection [24].

Machine learning to predict tactics was first considered by Gauthier et al. [14] in the context of the HOL4 theorem prover. His later improvements [15] added Monte-Carlo tree search, tactic orthogonalization, and integration of both Metis and a hammer [13]. A similar system for HOL Light was developed by Bansal et al. [1]. Nagashima and Kumar developed the proof search component [23] of such a system for Isabelle/HOL. This work builds upon Tactician [5,4], adapting and improving these works for dependent type theory and the Coq proof assistant.

6 Conclusion

We have implemented several new methods for learning tactical guidance of Coq proofs in the Tactician system. This includes better proof state features and an improved version of approximate k-nearest neighbor based on locality sensitive hashing forests. A completely new addition is our online implementation of random forest in Coq, which can now be used instead of or together with the k-nearest neighbor. We have also started to experiment with strong state-of-the-art learners based on gradient boosted trees, so far in an offline setting using binary learning with negative examples.

Our random forest improves very significantly on the k-nearest neighbor in an offline accuracy-based evaluation. In an online theorem-proving evaluation, the improvement is not as big, possibly due to the speed of the two methods and the importance of backtracking during the proof search. The methods are, however, quite complementary and running both of them in parallel increases the overall performance of Tactician from 33.7% (k-NN with the old features) to 40.4% in 40s. Our best new method (RF with the new features) now solves 36.2% of the problems in 40s.

The offline experiments with gradient boosted trees are so far inconclusive. They outperform k-nearest neighbor in top-10 accuracy, but the difference is small, and the random forest performs much better in this metric. Since the random forest learns only from positive examples, this likely shows that learning in the binary setting with negative examples is challenging on our Tactician data. In particular, we likely need good semantic feature characterizations of the tactics, obtained e.g., by computing the difference between the features of the proof states before and after the tactic application. The experiments, however, already confirm the importance of choosing good negative data to learn from in the binary setting.

References

1. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C., Wilcox, S.: Holist: An environment for machine learning of higher order logic theorem proving. In: Chaudhuri, K., Salakhutdinov, R. (eds.) Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. Proceedings of Machine Learning Research, vol. 97, pp. 454–463. PMLR (2019)
2. Bawa, M., Condie, T., Ganesan, P.: LSH Forest: Self-tuning indexes for similarity search. In: Ellis, A., Hagino, T. (eds.) Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005. pp. 651–660. ACM (2005)
3. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9), 509–517 (1975)
4. Blaauwbroek, L., Urban, J., Geuvers, H.: Tactic learning and proving for the Coq proof assistant. In: Albert, E., Kovács, L. (eds.) Proceedings of the 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR 2020. EPiC Series in Computing, vol. 73, pp. 138–150. EasyChair (2020)
5. Blaauwbroek, L., Urban, J., Geuvers, H.: The Tactician – A seamless, interactive tactic learner and prover for Coq. In: Benzmüller, C., Miller, B.R. (eds.) Proceedings of the 13th International Conference on Intelligent Computer Mathematics CICM 2020, Bertinoro, Italy, July 26-31, 2020. LNCS, vol. 12236, pp. 271–277. Springer (2020)
6. Breiman, L.: Random forests. Mach. Learn. **45**(1), 5–32 (2001)
7. Broder, A.Z.: On the resemblance and containment of documents. In: Carpentieri, B., Santis, A.D., Vaccaro, U., Storer, J.A. (eds.) Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings. pp. 21–29. IEEE (1997)
8. Chen, T., Guestrin, C.: XGBoost: A scalable tree boosting system. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 785–794 (2016)
9. Chvalovský, K., Jakubuv, J., Suda, M., Urban, J.: ENIGMA-NG: Efficient neural and gradient-boosted inference guidance for E. In: Fontaine, P. (ed.) Proceedings of the 27th International Conference on Automated Deduction, CADE 27, Natal, Brazil, August 27-30, 2019. LNCS, vol. 11716, pp. 197–215. Springer (2019)
10. Domingos, P.M., Hulten, G.: Mining high-speed data streams. In: Ramakrishnan, R., Stolfo, S.J., Bayardo, R.J., Parsa, I. (eds.) Proceedings of the sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. pp. 71–80. ACM (2000)

11. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E.: Making data structures persistent. J. Comput. Syst. Sci. **38**(1), 86–124 (1989)
12. Färber, M., Kaliszyk, C.: Random forests for premise selection. In: Lutz, C., Ranise, S. (eds.) Proceedings of Frontiers of Combining Systems (FroCoS'15). LNCS, vol. 9322, pp. 325–340. Springer (2015)
13. Gauthier, T., Kaliszyk, C.: Premise selection and external provers for HOL4. In: Leroy, X., Tiu, A. (eds.) Proceedings of the 4th Conference on Certified Programs and Proofs (CPP'15). pp. 49–57. ACM (2015)
14. Gauthier, T., Kaliszyk, C., Urban, J.: TacticToe: Learning to reason with HOL4 tactics. In: Eiter, T., Sands, D. (eds.) Proceedings of the 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, LPAR-21. EPiC Series in Computing, vol. 46, pp. 125–143. EasyChair (2017)
15. Gauthier, T., Kaliszyk, C., Urban, J., Kumar, R., Norrish, M.: TacticToe: Learning to prove with tactics. J. Autom. Reason. **65**(2), 257–286 (2021)
16. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Atkinson, M.P., Orlowska, M.E., Valduriez, P., Zdonik, S.B., Brodie, M.L. (eds.) Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99, September 7-10, 1999, Edinburgh, Scotland, UK. pp. 518–529. Morgan Kaufmann (1999)
17. Har-Peled, S., Indyk, P., Motwani, R.: Approximate nearest neighbor: Towards removing the curse of dimensionality. Theory Comput. **8**(1), 321–350 (2012)
18. Jones, K.S.: A statistical interpretation of term specificity and its application in retrieval. Journal of Documentation **60**(5), 493–502 (2004)
19. Kaliszyk, C., Urban, J., Michalewski, H., Olšák, M.: Reinforcement learning of theorem proving. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 31, pp. 8836–8847. Curran Associates, Inc. (2018)
20. Kaliszyk, C., Urban, J., Vyskočil, J.: Efficient semantic features for automated reasoning over large theories. In: Yang, Q., Wooldridge, M. (eds.) Proceedings of the 24th International Joint Conference on Artificial Intelligence, (IJCAI'15). pp. 3084–3090. AAAI Press (2015)
21. Mitchell, T.M.: Machine learning, International Edition. McGraw-Hill Series in Computer Science, McGraw-Hill (1997)
22. Nagashima, Y., He, Y.: Pamper: proof method recommendation system for isabelle/hol. In: Huchard, M., Kästner, C., Fraser, G. (eds.) Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 362–372. ACM (2018)
23. Nagashima, Y., Kumar, R.: A proof strategy language and proof script generation for Isabelle/HOL. In: de Moura, L. (ed.) Proceedings of the 26th International Conference on Automated Deduction, CADE 26. LNCS, vol. 10395, pp. 528–545. Springer (2017)
24. Piotrowski, B., Urban, J.: ATPboost: Learning premise selection in binary setting with ATP feedback. In: IJCAR. LNCS, vol. 10900, pp. 566–574 (2018)
25. Saffari, A., Leistner, C., Santner, J., Godec, M., Bischof, H.: On-line random forests. In: 12th IEEE International Conference on Computer Vision Workshops, ICCV Workshops 2009, Kyoto, Japan, September 27 - October 4, 2009. pp. 1393–1400. IEEE Computer Society (2009)
26. The Coq Development Team: The Coq proof assistant, version 8.11.0 (Oct 2019)
27. Zhang, C., Zhang, Y., Shi, X., Almpanidis, G., Fan, G., Shen, X.: On incremental learning for gradient boosting decision trees. Neural Process. Lett. **50**(1), 957–987 (2019)