Random Forests for Premise Selection

Michael Färber and Cezary Kaliszyk
{michael.faerber,cezary.kaliszyk}@uibk.ac.at

University of Innsbruck, Austria

Abstract The success rates of automated theorem provers in large theories highly depend on the choice of given facts. Premise selection is the task of choosing a subset of given facts, which is most likely to lead to a successful automated deduction proof of a given conjecture. Premise selection can be viewed as a multi-label classification problem, where machine learning from related proofs turns out to currently be the most successful method. Random forests are a machine learning technique known to perform especially well on large datasets. In this paper, we evaluate random forest algorithms for premise selection. To deal with the specifics of automated reasoning, we propose a number of extensions to random forests, such as incremental learning, multi-path querying, depth weighting, feature IDF (inverse document frequency), and integration of secondary classifiers in the tree leaves. With these extensions, we improve on the *k*-nearest neighbour algorithm both in terms of prediction quality and ATP performance.

1 Introduction

An increasing number of interactive theorem provers (ITPs) provide proof automation based on translation to automated theorem provers (ATPs): A user given conjecture together with a set of known facts in a more complicated logic of the ITP is translated to the logic of an ATP. If a proof is found by the ATP, it can be used to prove the conjecture in the ITP either by providing a precise small set of facts sufficient to prove the conjecture or the ATP proof can be used to recreate a skeleton of an ITP proof. To increase the success rate of the procedure, it is useful to identify a subset of theorems[1] that is most likely to produce a proof. This process is called *premise selection* (or *relevance filtering*) and is used in most ATP translation tools [AHK$^+$14], e.g. Sledgehammer/MaSh [KBKU13] for Isabelle/HOL [NPW02], or HOL(y)Hammer [KU15] for HOL Light [Har96], or Miz$\mathbb{AR}$ [KU13a] for Mizar [NK09].

Premise selection is also used in ATPs, for example the Sumo Inference Engine (SInE) [HV11] improves the prediction quality of the Vampire theorem prover [KV13] when working with large theories and its algorithm has also been implemented as a part of E-Prover [Sch13]. Nonetheless, as the complexity of

[1] As in premise selection we do not distinguish between axioms and lemmas, we denote their union as *theorems*. Furthermore, we denote the theorems used in a proof attempt as *premises*.

the translations to ATP highly depends on the lemmas to be translated, often only a subset of the lemmas is translated: For example in higher-order logic, if a constant f is always used with the same arity, e.g. $f(a, b)$ and $f(c, a)$, it can be directly translated as FOL function $f(x, y)$. However, if f appears with different arities, e.g. in $f(a)$ and $f(a, b)$, f cannot be translated as FOL function, and apply functors are necessary. Similarly, if a polymorphic constant only appears fully instantiated, its translation can be a FOL constant rather than a FOL function. Furthermore, the success rates of the ATPs depend significantly on the translation applied [BBP11], so avoiding unnecessary lemmas can shorten proof time by a better than linear factor. Premise selection for automated reasoning in ITPs is also different from that in ATPs due to a large knowledge base of previously proven theorems. The dependencies extracted both from ITP and ATP proofs can be used to further enhance premise selection.

Many algorithms used for premise selection stem from machine learning. To the best of our knowledge, one popular machine learning algorithm not yet tried in premise selection are random forests. In this paper we evaluate offline and online random forests for premise selection and propose a number of extensions to random forests that improve final ATP performance. Specifically we:

- investigate offline [AGPV13] and online [SLS^{+}09] random forests for premise selection,
- improve an offline random forest algorithm with incremental learning,
- add multi-path querying and depth weighting to improve multi-label output,
- integrate k-NN in the leaves of the random forest trees,
- evaluate the proposed extensions experimentally, confirming that random forests offer better prediction quality than previously used algorithms, and more theorems can be proven automatically by the ATPs.

Related work The Meng-Paulson relevance filter (MePo) [MP06] integrated in Isabelle/HOL as part of Sledgehammer was one of the first premise selectors for ITPs. It is an iterative algorithm, which counts function symbols in clauses and compares them to the function symbols in the conjecture to prove. In contrast to many other premise selectors, MePo does not consider the dependencies used to prove similar theorems.

Naive Bayes as implemented by the SNoW framework [CCRR99] was the first machine learning algorithm used in an automated reasoning loop, and thanks to dependencies, the prediction quality improved upon syntactic tools [Urb04]. Simple Perceptron networks have also been evaluated for HOL(y)Hammer predictions [KU14], and their results are weak but complementary to other methods.

Machine learning algorithms such as k-nearest neighbours [ZZ05] and Naive Bayes were integrated into Sledgehammer as part of MaSh (Machine learning for Sledgehammer) [KBKU13], significantly improving ATP performance on the translated problems. The single most powerful method used for premise selection in HOL(y)Hammer, Miz$\mathbb{AR}$, and Sledgehammer/MaSh is a customized implementation of k-NN [KU13b]. Stronger machine-learning methods that use kernel-based multi-output ranking (MOR [AHK^{+}14] and MOR-CG [Kü14]) were

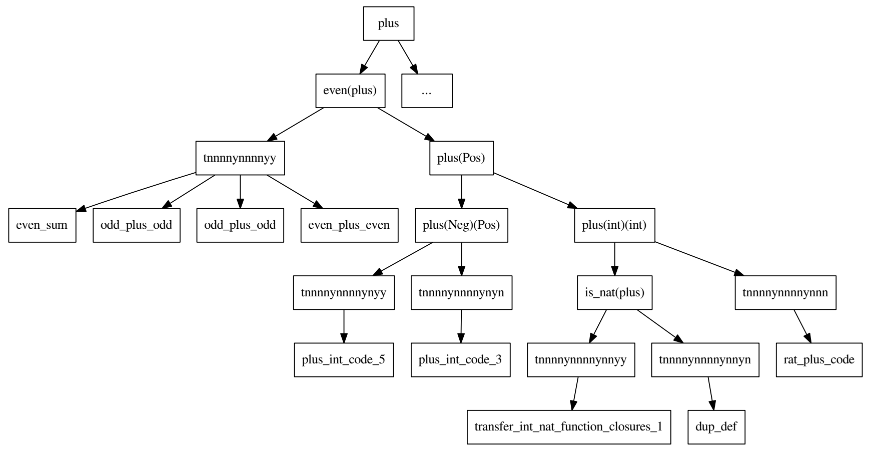

Figure 1: Excerpt from a decision tree trained on the Isabelle dataset. Leaf nodes have unique identifiers `t[yn]*`, which encode their position in the tree. The branch node with feature `even(plus)` has a positive leaf node with four theorems, namely `even_sum`, `odd_plus_odd` (two times), and `odd_plus_even` – all having features `plus` and `even(plus)`. The theorem `plus_int_code_3` has features `plus` and `plus(Pos)`, but neither `even(plus)` nor `plus(Neg)(Pos)`.

found to perform better, but were too slow to be of practical use for premise selection in large theories so far.

Decision trees are another machine learning method that can be used for premise selection: A binary decision tree is either a leaf $L(S)$ with data S or a branch $B(l, f, r)$ with a criterion (also called feature) f and two subtrees l and r. Querying a branch $B(l, f, r)$ involves querying l if the criterion f is fulfilled, otherwise querying r. Querying a leaf $L(S)$ returns S. A part of an example tree used in premise selection is shown in figure 1: Here, a criterion is the presence of certain symbols in a theorem, such as `plus`, and the data in the leaves are theorems that are relevant if the tree path to them corresponds to the symbols of the conjecture we seek to prove. We explain building and querying of decision trees in more detail in sections 3 and 4.

Random forests [Bre01] are a family of bagging algorithms [Bre96] known for fast prediction speed and high prediction quality for many domains [CNm06]. Many different versions of random forests [AGPV13,Bre96,LRT14,SLS$^+$09] have been proposed. In general, a random forest chooses random subsets of data to build independent decision trees, whose combined predictions form the prediction of the forest. Random forests are used in applications where large amounts of data needs to be classified in a short time, such as the automated proposal of advertisement keywords for web pages [AGPV13] or prediction of object positions in real-time computer graphics [SLS$^+$09].

2 Premise Selection

The goal of premise selection (sometimes also referred to as *relevance filtering*) is: Given a set of theorems T (i.e. a theorem corpus) and a conjecture c, find a set of premises $P \subseteq T$ such that an ATP is likely to find a proof of $P \vdash c$ [AHK$^+$14].

To find relevant premises, one can use information from previous proofs which premises were used to prove conjectures. We found that the following informal assumptions can be used to build fairly accurate premise selectors, when theorems are suitably characterised by features:

– Theorems sharing many features or rare features are similar.
– Theorems are likely to have similar theorems as premises.
– Similar theorems are likely to have similar premises.
– The fewer premises a theorem has, the more important they are.

The above assumptions can be encoded as a multi-label classification problem in machine learning. First we encode a given theorem corpus T as machine learning input: Every proven theorem $s \in T$ gives rise to a training *sample* $\langle s, \varphi(s), \lambda(s)\rangle$, which consists of the theorem s, the set of *features* $\varphi(s)$ and the set of *labels* $\lambda(s)$. The labels are the premises that were used to prove s.

The features $\varphi(s)$ are a characterisation of a theorem s. For example we can choose to characterise theorems by the constants and types present in their statements. The features of a set of samples S are $\varphi(S) := \bigcup_{s \in S} \varphi(s)$. We define those samples of S having or not having a certain feature f as

$$\begin{aligned} S_f &:= \{s \mid f \in \varphi(s)\}, \\ S_{\neg f} &:= S \backslash S_f . \end{aligned}$$

Example 1. The sample corresponding to the HOL Light theorem `ADD_SYM` stating $\vdash \forall m\, n.\, m + n = n + m$ is $\langle \texttt{ADD\_SYM}, \varphi(\texttt{ADD\_SYM}), \lambda(\texttt{ADD\_SYM})\rangle$ with:

$$\varphi(\texttt{ADD\_SYM}) = \{+, =, \forall, \texttt{num}, \texttt{bool}\}$$
$$\lambda(\texttt{ADD\_SYM}) = \{\texttt{ADD\_CLAUSES}, \texttt{ADD}, \texttt{ADD\_SUC}, \texttt{REFL\_CLAUSE}, \texttt{FORALL\_SIMP}, \texttt{num\_INDUCTION}\}$$

Samples encode the relationship between features and labels, i.e. which features occur in conjunction with which labels, both of which can be represented internally as sparse vectors. With this representation, we can view premise selection as an instance of a multi-label classification problem [TK07].

Definition 1 (Multi-label classifier). *Given a set of samples S, a multi-label classifier trained on S is a function r that takes a set of features φ and returns a list of labels $[l_1, \ldots, l_n]$ sorted by decreasing relevance for φ.*

Using multi-label classification, we can obtain suitable premises from a set of theorems S for a conjecture c as follows:

1. Obtain a multi-label classifier r for S.
2. Compute $\varphi(c)$, the features of the conjecture.
3. Return $r(\varphi(c))$, the list of labels predicted by the classifier.

2.1 Quality measures

To evaluate the quality of predicted premises, we can compare them to the actual premises from our training samples. We first introduce a notation: Given a sequence of distinct elements $X = [x_1, \ldots, x_n]$, we denote $X_i^e = [x_i, x_{i+1} \ldots, x_e]$. Furthermore, when it is clear from the context, we treat sequences as sets, where the set elements are the elements of the sequence.

The first quality measure is n-Precision, which is similar to Precision [Sor10], but considers only the first n predictions. It computes the percentage of premises from the training sample appearing among the first n predicted premises, which corresponds to our passing only a fixed maximal number of premises to ATPs. If not stated otherwise, we use 100-Precision in our evaluations.

Definition 2 (n-Precision). *n-Precision for a sequence of predictions P and a set of labels L is*

$$\mathrm{Prec}_n(P, L) = \frac{|L \cap P_1^n|}{|L|}.$$

The second measure, AUC, models the probability that for a randomly drawn label $l \in L$ and a randomly drawn label $m \notin L$, l appears in the predictions before m.

Definition 3 (AUC [Faw04]). *Given a sequence of predictions P and a set of labels L, the area under ROC curve (AUC) for the predictions is*

$$\mathrm{AUC}(P, L) = \begin{cases} \frac{\sum_{n=1}^{|P|} |L \cap P_1^n|}{|L| \cdot |P \backslash L|} & \textit{if } |L| \cdot |P \backslash L| > 0 \\ 1 & \textit{if } |L| \cdot |P \backslash L| = 0. \end{cases}$$

2.2 Evaluation

We now explain how to evaluate predictor performance on a set of samples. For this, we define a subset of the samples as *evaluation samples*, for which the classifier will predict premises by iterating over all samples in order and predicting $\lambda(e)$ for each evaluation sample e before learning e, as illustrated in figure 2. We can evaluate the quality of the predictions in two ways: First, they can be compared to the actual labels of the evaluation samples, using the a quality measure from section 2.1. Second, the predictions can be translated to an ATP problem and given to an automated prover.

2.3 Used datasets

We use the Mizar MPTP2078 dataset [AHK$^+$14] updated to Mizar 8.1.02 [KU13a] using α-normalised subterms as features, the Isabelle 2014 theory HOL/Probability together with its dependencies [KBKU13], and the core library of HOL Light SVN version 193 [KU15]. The statistics are shown in table 1.

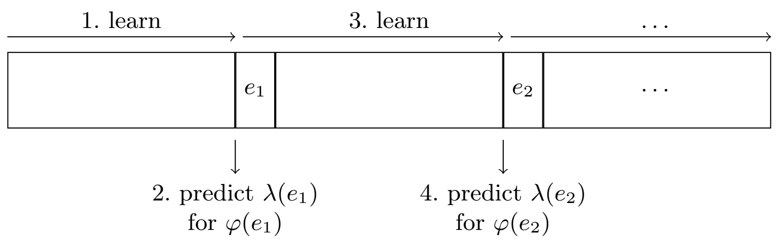
1. learn
3. learn
. . .
e_1
e_2
. . .
2. predict $\lambda(e_1)$ for $\varphi(e_1)$
4. predict $\lambda(e_2)$ for $\varphi(e_2)$

Figure 2: In an evaluation, an arbitrary number of samples is learned in a white block until an evaluation sample e is encountered, for which labels $\lambda(e)$ are predicted.

Dataset	Samples	Evaluation samples	Features	Avg. labels per sample	Avg. features per sample
Mizar	3221	2050	3773	8.8	14.2
HOL Light	2548	2247	4331	2.6	13.4
Isabelle	23442	1656	31308	4.2	23.1

Table 1: Datasets used in the evaluation.

3 Existing algorithms

In this section we describe offline and online random forests and evaluate them in the context of ITP premise selection.

Multi-Label Learning with Millions of Labels Agrawal et al. [AGPV13] use random forests to learn large amounts of data, in order to obtain relevant advertising keywords for web pages. Their algorithm builds several decision trees on random subsets of the data as follows: Given a set of samples S to learn and the minimal number of samples μ which a leaf has to contain (we describe this in section 4.3), it returns a decision tree. The algorithm first determines a splitting feature (explained in section 4.4) for S, which is a feature f that splits S in two sets S_f (samples having f) and $S_{\neg f}$ (samples not having f). If $|S_f| < \mu$ or $|S_{\neg f}| < \mu$, the algorithm returns a leaf node containing S, otherwise the algorithm recursively calculates subtrees for S_f and $S_{\neg f}$ and combines them into a branch node with the splitting feature f.

This approach has several disadvantages when used for premise selection: While we need to learn data quickly and query only a few times after each learning phase, the algorithm of Agrawal is optimised to answer queries in logarithmic time, whereas its learning phase is relatively slow. Furthermore, the algorithm is an *offline* algorithm, meaning that in order to learn new samples, it is necessary to rebuild all trees. We found that our implementation of this method was several magnitudes slower than k-NN even for small datasets, rendering it impracticable for incremental learning. Furthermore, the prediction quality was lower than expected: For the first 200 evaluation samples of the Mizar dataset, a random forest with 4 trees and 16 random features evaluated at every tree branch achieved an AUC of 82.96% in 1m22sec, whereas k-NN achieved an AUC of 95.84% in 0.36sec. In section 4, we show how to improve the prediction quality and speed of this algorithm for premise selection.

On-line Random Forests Saffari et al. [SLS$^+$09] present an online random forest algorithm, in which all trees in the forest are initially leaf nodes. When learning a new sample, it is added to all trees with a probability determined by a Poisson distribution with $\lambda = 1$ [OR01]. Adding a sample to a leaf node consists of adding the sample to the samples in the leaf node. As soon as the number of samples in a leaf node exceeds a certain threshold or a sufficiently good splitting feature for the sample set is found, the leaf node splits into a feature node and two leaf nodes. When adding a sample to a feature node, the sample gets added to the left or to the right child of the node, according to whether or not it has the node's feature.

The method introduces a bias in that features which appear in early learned samples will be at the tree roots. Saffari et al. solve this problem by calculating the quality of predictions from each tree (OOBE, out-of-bag error) and by periodically removing trees with a high OOBE. However, this introduces a bias towards the latest learned samples, which is useful for computer graphics applications such as object tracking, but undesirable for premise selection, as the advice asked from a predictor will frequently not correspond to the last learned theorems. Therefore, we do not use the approach of [SLS$^+$09], but adapt its use of probability distributions to create online versions of bagging algorithms in section 4.2.

4 Adaptations to Random Forests for Premise Selection

In this section, we describe the changes we made to the algorithms described in section 3 to obtain better results for premise selection.

4.1 Sample selection

When learning new samples S, one needs to determine which trees learn which samples. In [AGPV13], each tree in a forest randomly draws n samples from S. This approach may introduce a bias, namely that some samples are drawn more often than others, while some samples might not be drawn (and learned) at all. Therefore, instead of each tree drawing a fixed number of samples to learn, in our approach, each sample draws a fixed number of trees by which it will be learned, where we call this fixed number *sample frequency*. This approach has the advantage that by definition, every sample is guaranteed to be learned as often as all other samples.

4.2 Incremental update

We present two methods to efficiently update random forests incrementally: The first one is a method applicable to all kinds of classifiers, the second one is an optimised update procedure for decision trees.

Onlining bagging algorithms Given a bagging algorithm (such as random forests) whose individual predictors (in our scenario the decision trees of the forest) learn a random subset of samples offline, we show a method for decreasing the runtime of learning new data incrementally. The method is based on the observation that, when learning only a small number of new samples (compared to the number of samples already learned), most predictors will not include any of those new samples, thus they do not need to be updated. To model this, let r be a binomially distributed random variable $r \sim B(s, P)$, where s is the number of samples in each predictor and $P = \frac{n_{new}}{n_{new}+n_{old}}$ is the probability of drawing a new sample from the common pool of new and old samples. r then models the number of new samples drawn by a predictor. Each predictor evaluates the random variable r, and if its value r_p is 0, the predictor can remain unchanged. Only if r_p is greater than 0, the predictor is retrained with r_p samples from the set of new samples and $s - r_p$ samples from the set of old samples.

While this method gives a performance increase over always rebuilding all predictors, it still frequently retrains whole predictors. As training a decision tree is a very expensive operation, this method is clearly suboptimal for our setting, therefore we present a method to update trees efficiently in the next section.

Tree update We show an improved version of the first algorithm given in section 3, which updates trees with new samples. Given a tree t and a set of new samples S, the algorithm calculates S', which is the union of S with all the samples in the leaf nodes of t, and a splitting feature f for S'. If t is a node with f as a splitting feature, we recursively update both subtrees of t with S_f and $S_{\neg f}$ respectively. Otherwise, we construct a new tree for S': If $|S'_f| < \mu$ or $|S'_{\neg f}| < \mu$, we return a leaf node with S', otherwise we construct subtrees for S'_f and $S'_{\neg f}$ and return a branch node with f as splitting feature.

This algorithm returns the same trees as the original algorithm, but can be significantly faster in case of updates; for example, predicting advice for the whole Mizar dataset takes 21m27sec with this optimisation and 57m22sec without.

4.3 Tree size

At each step of the tree construction, the given set of samples S is split in two by a splitting feature. A leaf containing S is created if one of the two resulting sets contains fewer samples than the minimum number of samples μ. We evaluated three functions to calculate μ, which depend on the samples of the whole tree, namely $\mu_{\log}(S) = \log |S|$, $\mu_{\text{sqrt}}(S) = \sqrt{|S|}$, and $\mu_{\text{const}}(S) = 1$. In [AGPV13] only $\mu_{\log}$ is used.

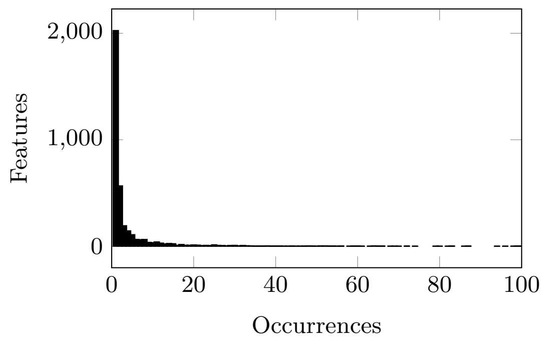

Figure 3: Feature histogram for the Mizar MPTP2078 dataset [AHK$^+$14]. For example, there are 2026 features which occur only a single time among all samples, and only 34 that occur ten times.

4.4 Feature selection

We determine a splitting feature for a set of samples S in two steps: First, one selects a set of features $F \subseteq \varphi(S)$ to evaluate, then, one evaluates each of the features in F to obtain a suitable splitting feature.

Obtaining evaluation features In [AGPV13], the evaluation features are obtained by randomly drawing with replacement (meaning you draw an element from a set, then place it back in the set) a set of features φ_R from $\varphi(S)$, where $n_R = |\varphi_R|$ is a user-defined constant. When we applied the method in the context of premise selection, we frequently obtained trees of small height with many labels at each leaf, because many features occur relatively rarely in our datasets, see figure 3. Taking larger subsets of random features alleviates this problem, but it also makes the evaluation of the features slower. To increase performance, we determine for each feature in φ_R how evenly it divides the set of samples in two, by evaluating

$$\sigma(S, f) := \frac{||S_f| - |S_{\neg f}||}{|S|}.$$

The best output of $\sigma(S, f)$ for a feature is 0, which is the case when a feature splits the sample set S in two sets of exactly the same size, and the worst output is 1, when the feature appears either in all samples or in none. In the evaluation phase, we consider only n_σ features φ_σ of φ_R that yield the best values for $\sigma(S, f)$. The motivation behind this is to preselect features which are more likely candidates to become splitting features, thus saving time in the evaluation phase.

Evaluating features The best splitting feature for a set of samples S should be a feature f which makes the samples in S_f and $S_{\neg f}$ more homogenous compared to S [AGPV13]. Common measures to determine splitting features are information gain and Gini impurity [RS04]. Furthermore, to obtain a tree that is not too high, it is desirable for a splitting feature to split S evenly, such that S_f and $S_{\neg f}$ have roughly the same number of labels.

In general, we look for a function $s(S, f)$, which determines the quality of f being a splitting feature for S. The best splitting feature can then be obtained by $\arg\min_{f \in \varphi_\sigma} s(S, f)$. We evaluated two concrete implementations for $s(S, f)$:

1. $\sigma(S, f)$: While σ optimally divides S into two evenly sized sets S_f and $S_{\neg f}$, it does not take into account their homogenicity.
2. $G(S, f) = \frac{1}{|S|}\left(|S_f| g(S_f) + |S_{\neg f}| g(S_{\neg f})\right)$: The Gini impurity [AGPV13] g measures the frequency of each label among a set of samples, and gives labels with very high or very low frequency a low value. That means that the more similar the samples are (meaning they possess similar labels), the lower the Gini impurity.

Definition 4 (Gini impurity). *Gini impurity $g(S)$ of a set of samples S is*

$$g(S) = \sum_{l \in \lambda(S)} p_S(l)\,(1 - p_S(l))$$

$$p_S(l) = \sum_{s \in S} p_S(l|s) p(s), \quad p_S(l|s) = \frac{|\lambda(s) \cap \{l\}\,|}{|\lambda(s)|}, \quad p_S(s) = \frac{|\lambda(s)|}{\sum_{s' \in S} |\lambda(s')|}$$

4.5 Querying a tree

Querying a tree with features F corresponds to finding samples S from the tree maximising $P(S|F)$. We show a multi-path querying algorithm, as well as a method to obtain labels from the samples with classifiers such as k-NN.

Multi-path querying To query a decision tree with features F, a common approach is to recursively go to the left subtree l of a branch node $B(l, f, r)$ if $f \in F$ and to the right if not, until encountering a leaf $L(S)$, upon which one returns S. We found that this approach frequently missed samples with interesting features when these did not completely correspond to the features we queried for. This is why we considered a different kind of tree query, which we call *multi-path querying* (MPQ) in contrast to *single-path querying* (SPQ). MPQ considers not only the path with 100% matching features, but also all other paths in the tree. At each branch node where the taken path differs from that foreseen by the splitting feature of the node, we store the depth d of the node, as illustrated in figure 4. The output of a multi-path query for a tree t and features F is $mq_F(t, 0, \emptyset)$, defined as follows:

$$mq_F(t, d, E) = \begin{cases} (S, d, E) & t = L(S) \\ mq_F(l, d+1, E) \cup mq_F(r, d+1, E \cup \{d\}) & t = B(l, f, r) \wedge f \in F \\ mq_F(r, d+1, E) \cup mq_F(l, d+1, E \cup \{d\}) & t = B(l, f, r) \wedge f \notin F \end{cases}$$

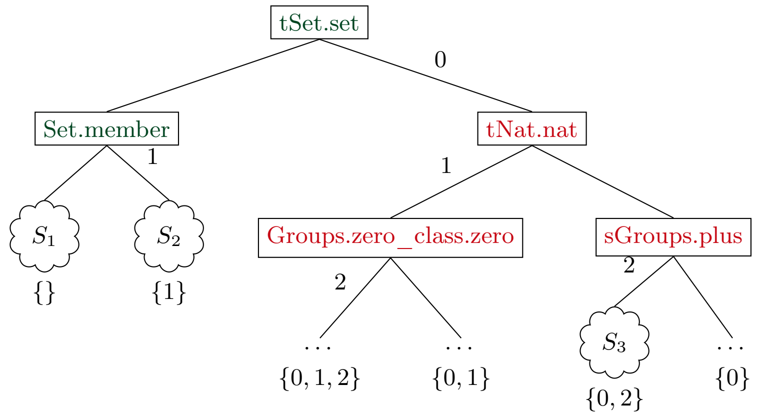

Figure 4: Multi-path query example, where the tree is an excerpt from an actual random forest tree generated from the Isabelle/HOL Probability dataset. Query features are {tSet.set, Set.member}. The numbers next to the branches indicate the depth of wrongly taken decisions, which are accumulated and shown below the samples at the bottom.

Depth weighting We want to assign to each tree leaf a weight, which indicates how well the features F correspond to the features along the path from the root of the tree to the leaf. To do this, we consider the depths of the branch nodes where we took a different path than foreseen by F, and calculate for each of the depths a weight, which we later combine to form a branch or sample weight.

For each $e \in E$, where $(S, d, E) \in mq_F(t, 0, \emptyset)$, we calculate a depth weight, where the constant μ represents the minimal weight: $e_{\text{ascending}}(d, e) = \mu + (1 - \mu)\left(\frac{e}{d}\right)$, $e_{\text{descending}}(d, e) = 1 - (1-\mu)\left(\frac{e}{d}\right)$, $e_{\text{inverse}}(d, e) = 1 - \frac{1-\mu}{e+1}$, and $e_{\text{const}}(d, e) = \mu$. Using the depth weights, we calculate a weight for each sample:

$$w_t(s) = \sum_{(S,d,E) \in mq_F(t,0,\emptyset),\, s \in S} \prod_{e \in E} e_i(d, e).$$

Classifier in leaves Regular decision trees with single-path querying return all the labels of the chosen branch. To order the results from multiple branches in a tree, which is necessary with multi-path querying, we run a secondary classifier on all the leaf samples of the tree. The secondary classifier is modified to take into account the weight of each branch. In our experiments, the secondary classifier is a k-NN algorithm adapted for premise selection (IDF, premise relevance inversely proportional to the number of premises [KU13b]), which we modified to accept sample weights: k-NN will give premises that appear in samples with higher weights precedence over those from samples with lower weights. In default k-NN, all samples would have weight 1, while in our secondary classifier the weight of a sample s is given by $w_t(s)$, which stems from the path to s in the decision tree.

4.6 Querying a forest

We query a forest with a set of features F by querying each tree in the forest with F, combining the prediction sequences $\overrightarrow{L}$ of all trees. For each label l, we calculate its rank in a prediction sequence $L = [l_1, \dots, l_n]$ as:

$$\varrho(l,\, L) = \begin{cases} i & \text{if } l = l_i \text{ and } l_i \in L \\ m & \text{otherwise} \end{cases}$$

Here, m is a maximal rank attributed to labels that do not appear in a prediction sequence. Then, for each label, we calculate its ranks $R(l) = \biguplus_{L \in \overrightarrow{L}} \varrho(l, L)$ for all prediction sequences. We sort the labels by the arithmetic, quadratic, geometric, or the harmonic mean of $R(l)$ in descending order to obtain the final prediction sequence.

5 Experiments

We implemented the algorithms from section 4 in Haskell.[2] Our experimental results for the Mizar dataset are given in table 2: Random forests give best results when combined with multi-path querying and path-weighted k-NN+IDF classifier in the leaves. Both considering Gini impurity and taking random subsets of features decrease the prediction quality, while having a very negative impact on runtime. Different sample selection methods (samples draw trees vs. trees draw samples) have a large impact when using small sample frequencies, but when using higher sample frequencies, the difference is negligible. In this evaluation, we simulated single-path querying (SPQ) by a constant depth weight with $\mu = 0$ (meaning that all non-perfect tree branches receive the minimal score 0). Running this method takes longer than real SPQ, but gives a good upper bound on SPQ's prediction quality. Random forests have a longer runtime than k-NN, but still, the average prediction time for our test set is below one second, which is sufficient for premise selection.

To produce the number of proven theorems in table 3, we predict max. 128 (for Mizar, for HOL Light 1024) premises for each conjecture, translate the chosen facts (if no PS: all previous facts) together with the conjecture to TPTP first-order formulas [Sut09] and run E-Prover 1.8 [Sch13] using automatic strategy scheduling with 30 seconds timeout.

Alama et al. [AHK$^+$14] have reported 548 proven theorems with Vampire (timeout = 10s) without external premise selection, which their best premise selection method (MOR-40/100) increases to 824 theorems (+50.4%). On our data, E (timeout = 10s) without premise selection proves only 414 theorems, increasing with timeout = 30s to 653 theorems (+57.7%) and with timeout = 10s and RF premise selection to 962 (+132.3%).

In table 4, we compare ATP runtime required to prove the same number of

[2] Source and detailed statistics (also for HOL Light and Isabelle datasets) are available at `http://cl-informatik.uibk.ac.at/~mfaerber/predict.html`.

Configuration	100-Prec [%]	AUC [%]	Runtime [min]	Avg. time per prediction [s]
k-NN + IDF	87.5	95.39	**0.5**	**0.02**
RF (IDF)	88.0	95.68	32	0.93
RF (no IDF)	77.8	91.40	25	0.75
RF (single-path query)	53.7	60.86	37	1.07
RF (sample freq. = 2, trees draw s.)	65.6	72.76	2	0.05
RF (sample freq. = 2, samples draw t.)	88.0	95.59	4	0.10
RF (random features $n_R = 32$)	88.0	95.65	151	4.41
RF (Gini impurity, $n_\sigma = 2$)	88.0	95.65	97	2.84
RF (Gini impurity, $n_\sigma = 16$)	88.0	95.62	220	6.44
RF ($e_{\text{ascending}}$)	88.0	95.72	36	1.07
RF ($e_{\text{descending}}$)	88.1	95.66	39	1.15
RF (e_{inverse})	88.0	95.68	38	1.12
RF (e_{const})	88.1	95.81	37	1.08
RF (arithmetic mean)	87.5	95.49	33	0.98
RF (geometric mean)	88.0	95.67	35	1.01
RF (quadratic mean)	87.4	95.34	33	0.97
RF (100 trees, sample freq. = 50)	88.5	95.85	137	4.01
RF (24 trees, sample freq. = 12)	88.5	95.83	31	0.90
RF (24 trees, sample freq. = 12, e_{const})	**88.6**	**95.91**	22	0.66

Table 2: Results for Mizar dataset. By default, we use 4 trees with a sample frequency of 16, samples draw trees. The minimal sample function is $\mu_{\log}$, we do not use Gini impurity, and we use e_{Inverse} with $\mu = 0.8$. The final prediction is obtained by running k-NN with IDF over the weighted leaf samples of each tree, combining results with the harmonic mean.

	k-NN AUC	RF AUC	k-NN Prec	RF Prec	k-NN proved	RF proved	Total
Mizar	0.9539	0.9591	0.875	0.886	931	959 (+3.0%)	2050
HOL Light	0.9565	0.9629	0.919	0.929	789	823 (+4.3%)	2247

Table 3: Results of k-NN and random forest predictions for two different datasets. For random forests, we used the best configuration from table 2, i.e. 24 trees, sample frequency 12, and constant depth weight.

Classifier	Classifier runtime	E timeout	E runtime	Total runtime
k-NN	0.5min	15sec	341min	341min
RF	22min	10sec	252min	272min

Table 4: Comparison of runtime necessary to achieve the same number of proven theorems (969) for the Mizar dataset.

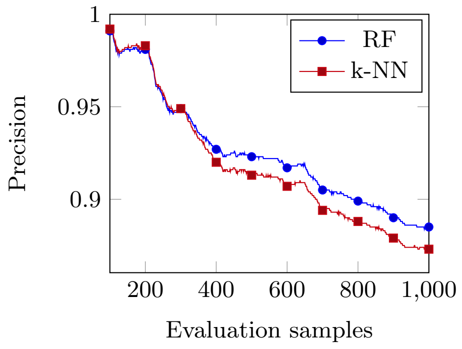

Figure 5: Comparison of k-NN with random forests by number of evaluation samples on Mizar dataset.

theorems using k-NN and RF predictions. While RF classification requires more runtime than k-NN, the ATP timeout can be decreased by more than 25%, resulting in overall runtime reduction of about 20%.

Number of evaluation samples In figure 5, we show how the prediction quality develops for the Mizar dataset as more data is learned: For this purpose, we calculated statistics for the predictions of just our first evaluation sample, then for the first two, etc. When comparing the output of our random forest predictor (24 trees, sample frequency 12, constant depth weight) with k-NN, we see that it consistently performs better.

6 Conclusion

We evaluated several random forest approaches for ATP premise selection: Without modifications, the algorithms return worse predictions than the current state-of-the-art premise selectors included in HOL(y)Hammer, Miz$\mathbb{AR}$, and Sledgehammer/MaSh, and the time needed to select facts from a larger database is significant. We then proposed a number of extensions to the random forest algorithms designed for premise selection, such as incremental learning, multi-path querying, and various heuristics for the choice of samples, features and size of the trees. We combined random forests with a k-NN predictor at the tree leaves of the forest, which increases the number of theorems from the HOL Light dataset that E-Prover can successfully reprove over the previous state-of-art classifier k-NN by 4.3%. We showed that to attain the same increase with k-NN, it is necessary to run E-Prover for 50% longer.

In scenarios where the number of queries is large in comparison with the number of learning phases, the random forest approach is an effective way of improving prediction quality while keeping runtime acceptable. This is the case for usage in systems such as HOL(y)Hammer and Miz$\mathbb{AR}$, but not for Sledgehammer, where data is relearned more frequently. The performance of random forests could still be improved by recalculating the best splitting feature only after having seen a certain minimal number of new samples since the last calculation of the best feature. This would improve learning speed while not greatly altering prediction results, because it is relatively unlikely that adding few samples to a big tree change the tree's best splitting feature. Further runtime improvements could be made by parallelising random forests.

Acknowledgements

We thank Josef Urban as well as anonymous CADE and FroCoS referees for their valuable comments. This work has been supported by the Austrian Science Fund (FWF) grant P26201.

References

AGPV13. R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. Multi-label learning with millions of labels: recommending advertiser bid phrases for web pages. In *Proceedings of the 22nd International Conference on World Wide Web*, WWW '13, pages 13–24, 2013.

AHK$^+$14. J. Alama, T. Heskes, D. Kühlwein, E. Tsivtsivadze, and J. Urban. Premise selection for mathematics by corpus analysis and kernel methods. *Journal of Automated Reasoning*, 52(2):191–213, 2014.

BBP11. J. C. Blanchette, S. Böhme, and L. C. Paulson. Extending Sledgehammer with SMT solvers. In *Automated Deduction – CADE-23*, volume 6803, pages 116–130. Springer, 2011.

Bre96. L. Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

Bre01. L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

CCRR99. A. J. Carlson, C. M. Cumby, J. L. Rosen, and D. Roth. SNoW user guide, 1999.

CNm06. R. Caruana and A. Niculescu-mizil. An empirical comparison of supervised learning algorithms. In *23rd Intl. Conf. Machine learning (ICML'06)*, pages 161–168, 2006.

Faw04. T. Fawcett. ROC graphs: Notes and practical considerations for researchers. Technical report, HP Laboratories, March 2004.

Har96. J. Harrison. HOL Light: A tutorial introduction. In *Formal Methods in Computer-Aided Design*, volume 1166, chapter Lecture Notes in Computer Science, pages 265–269. Springer, 1996.

HV11. K. Hoder and A. Voronkov. Sine qua non for large theory reasoning. In *Automated Deduction - CADE-23*, volume 6803, pages 299–314. Springer, 2011.

KBKU13. D. Kühlwein, J. C. Blanchette, C. Kaliszyk, and J. Urban. MaSh: Machine learning for Sledgehammer. In *Interactive Theorem Proving*, volume 7998, pages 35–50. Springer, 2013.

KU13a. C. Kaliszyk and J. Urban. MizAR 40 for Mizar 40. *CoRR*, 2013.

KU13b. C. Kaliszyk and J. Urban. Stronger automation for Flyspeck by feature weighting and strategy evolution. In *PxTP 2013*, volume 14 of *EPiC Series*, pages 87–95. EasyChair, 2013.

KU14. C. Kaliszyk and J. Urban. Learning-assisted automated reasoning with Flyspeck. *Journal of Automated Reasoning*, 53(2):173–213, 2014.

KU15. C. Kaliszyk and J. Urban. HOL(y)Hammer: Online ATP service for HOL Light. *Mathematics in Computer Science*, 9(1):5–22, 2015.

KV13. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In *Computer Aided Verification*, volume 8044, pages 1–35. Springer, 2013.

Kü14. D. A. Kühlwein. *Machine Learning for Automated Reasoning*. PhD thesis, Radboud Universiteit Nijmegen, April 2014.

LRT14. B. Lakshminarayanan, D. Roy, and Y. W. Teh. Mondrian forests: Efficient online random forests. In *Advances in Neural Information Processing Systems*, 2014.

MP06. J. Meng and L. C. Paulson. Lightweight relevance filtering for machine-generated resolution problems. In *ESCoR: Empirically Successful Computerized Reasoning*, pages 53–69, 2006.

NK09. A. Naumowicz and A. Kornilowicz. A brief overview of Mizar. In *22nd International Conference, TPHOLs 2009*, pages 67–72, 2009.

NPW02. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

OR01. N. C. Oza and S. J. Russell. Online bagging and boosting. In *Proceedings of the Eighth International Workshop on Artificial Intelligence and Statistics, AISTATS 2001, Key West, Florida, US, January 4-7, 2001*, 2001.

RS04. L. E. Raileanu and K. Stoffel. Theoretical comparison between the Gini index and information gain criteria. *Annals of Mathematics and Artificial Intelligence*, 41(1):77–93, 2004.

Sch13. S. Schulz. System description: E 1.8. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Proc. of the 19th LPAR, Stellenbosch*, volume 8312 of *LNCS*. Springer, 2013.

SLS$^+$09. A. Saffari, C. Leistner, J. Santner, M. Godec, and H. Bischof. On-line random forests. In *3rd IEEE ICCV Workshop on On-line Computer Vision*, 2009.

Sor10. M. S. Sorower. A literature survey on algorithms for multi-label learning. *Oregon State University, Corvallis*, 2010.

Sut09. G. Sutcliffe. The TPTP problem library and associated infrastructure. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

TK07. G. Tsoumakas and I. Katakis. Multi-label classification: An overview. *Int J Data Warehousing and Mining*, 2007:1–13, 2007.

Urb04. J. Urban. MPTP - motivation, implementation, first experiments. *J. Autom. Reasoning*, 33(3-4):319–339, 2004.

ZZ05. M.-L. Zhang and Z.-H. Zhou. A k-nearest neighbor based algorithm for multi-label classification. In *Proceedings of the 1st IEEE International Conference on Granular Computing (GrC'05)*, pages 718–721, Beijing, China, 2005.