Final Project of Machine Learning - Forests and Ferests

Jonas Uhrig
03616049
jonas.uhrig@in.tum.de

Michael Wolfram
03616011
michaelwolfram@gmx.de

Abstract

In this paper we describe our final project for the Machine Learning class of 2013/14 at TUM. We use data set 4) containing information about body postures and movements and apply two similar Machine Learning algorithms that were implemented within this project. The used algorithms are called 'Ferns' and 'Binary Tree' or 'Random Forest' - we also cover different variations of those algorithms. Moreover, we applied heuristical approaches used for Trees to our Ferns to evaluate advantages regarding classification accuracy.
The programming language of our choice is Matlab - some of the used approaches on Ferns are based on ideas from Mustafa Özuyal et al. [1], the development of the trees and forests was only based on ideas from the lecture.

1 Approach

We already know about Trees and Forests from the lecture - additionally we read of a Machine Learning algorithm called *Ferns* [1], which was recently (2007) introduced to detect objects or keypoints in images. Ferns have a very similar behaviour as Forests and, in fact, are completely replaceable - though the important difference lies in the striking speed for training a Fern. This is the case because a Fern has only n nodes where a Tree has 2^n nodes. A Tree can be transformed into a Fern by applying the same tests in every node of a Tree-layer and then 'shrinking' it horizontally to consist of only one node per layer.

For object detection purposes, Ferns are built using binary values as feature vector which initially posed a problem for the chosen PUC-Rio data set containing continuous (e.g the sensor data) and categorical (e.g gender, names) features. A key point of Ferns is chance so we thought it might serve to randomly choose a threshold within the range of each feature and do a simple comparison, resulting in a new binary feature value - this approach was also used by Kursa [2], which encouraged us to continue the implementation.

The goal of this project is, besides the implementation of the two chosen algorithms, the evaluation of Ferns on data with non-binary attributes. Within the original application area of Ferns (fast keypoint recognition), Ferns outperform Trees. Thus, we wanted to evaluate if this relation also holds true for the given data set. To achieve this, we applied various available methods to optimize the classification accuracy of our algorithms in order to compare Ferns and Trees, also regarding times for training and testing.

2 Data

The provided data set consists of 165633 samples of recorded body postures and movements. On the one hand the features contain information about the humans themselves: name, gender, age, height, weight and body-mass-index. On the other hand sensor readings of accelerometers from human joints are included. This results in a total of 18 features.
The possible postures of the subjects are: 'sitting', 'sittingdown', 'standing', 'standingup' and 'walking'. The distribution of the data set among these classes is as follows:

'sitting':	50631
'sittingdown':	11827
'standing':	47370
'standingup':	12415
'walking':	43390

Given that unequal distribution we selected not just a certain number but rather a particular percentage of randomly drawn data samples from each class for the testing set as stated in the project description. We used the usual partition of $\frac{2}{3}, \frac{1}{3}$ for the training and testing set.
During the evaluation of Forests we were faced with limitations regarding time and computation power. Therefore, we were forced to make a decision whether to evaluate Forests on the whole provided data set or to analyse Forests for a broader variety of combinations of their parameters. Since we consider the latter to be more important we decided to do the majority of our analysis of Forests on 10% of the data set.

3 Method

This section gives a short overview of the properties of both methods. The algorithms and basic methods to optimize them were taken from the original papers (see section "References") and some lecture slides. In the final implementation we made some slight changes to cope with the given restrictions of this project.

3.1 Forest

This section concisely describes key facts of our implementation of Forests.

3.1.1 Basic Principle

As known from the lecture, we implemented a Forest as ensemble of a certain number of Decision Trees. Our Trees are defined by the following parameters:
We implemented bagging and feature subset selection for the training of each Tree of a Forest to overcome overfitting on the whole data set. When using bagging, the data set that a single Tree may use for its training phase is randomly drawn from the original training set of the Forest with replacement. Therefore, pruning as third method became superfluous. The random selection of feature subsets induces a parameter to the Tree: the number $d < D$ of features, with D being the

total number of features, which is used for finding best splits in each node of the Tree. Other properties of a Tree are its criteria when to stop growing. We implemented the four suggestions from the lecture:

- The depth of the tree can be limited to a certain maximum
- A lower bound for the benefit of another split can be specified
- If the number of training samples in a certain node is below a given threshold it will not be split any more and is marked as leaf and lastly
- The tree stops growing at a particular node if the distribution of labels/classes of the training data in this node is pure

For the last stopping heuristic we implemented the impurity measures *Gini index*, *Entropy* and *misclassification rate* as stated in the lecture. After some small initial tests we decided to concentrate on the *Gini index* for further testing since it was more accurate on a subset of the given training data even though *Entropy* was slightly faster.
We used the same settings for each Tree of a Forest.

3.1.2 Optimizations

Due to the fact that we were faced with limitations regarding time and computation power we did minor modifications to the building process of a Forest. Rather than building a whole new Forest for each different number of Trees within the Forest we just added a new Tree to an existing Forest, trained that particular Tree - with bagging and feature subset selection - and then evaluated the 'updated' Forest. Thus, we were able to monitor the performance of a Forest with its given set of Trees after adding new Trees to it. To further improve the performance of our Trees we iterate over the whole predefined subset of feature dimensions but only over the unique feature values among one feature dimension when searching for the best split at a particular node. For the actual choice of parameters refer to the section "Results".

3.2 Ferest

In this section we describe Ferns as well as *Ferests*, which are forests of Ferns [1]. We describe methods for training and testing together with the mathematical background and then consider different adjustments that have been made for testing and optimization issues.

3.2.1 Mathematical Background

Generally, what training is supposed to achieve is a distribution over given feature vectors that classifies a new sample into one of the trained classes. In other words, we want to obtain the class which has the highest posterior probability over all class labels, given the specific features of a sample: $\arg\max_k P(C_k|f_1, f_2, \cdots, f_N)$. Applying Bayes' rule results in the following expression: $\arg\max_k P(f_1, f_2, \cdots, f_N|C_k)P(C_k)$, which is proportional to the last expression, meaning that if we find the joint distribution over all features, we know to which class a sample belongs to - unfortunately, this is very hard to get. Instead, we make the Naive Bayes assumption of independent and identically distributed features which leads us to the first classification formula:

$$Class(f) \equiv \arg\max_k P(C_k) \prod_{n=1}^{N} P(f_n|C_k)$$

The assumption that all features in our data set have nothing to do with each other is rather strong and often incorrect. Thus, this usually delivers quite accurate results and is easy and fast to learn.

The idea of a Ferest is to combine joint class-conditional distributions with a Naive assumption to a *Semi-Naive* method which achieves accurate results within a good range of complexity and computational effort. Instead of assuming conditional independence of features, a Ferest uses L randomly built Ferns F_l of size S and assumes independence between those. The new and final classification formula for a Ferest is then:

$$Class(f) \equiv \arg\max_k P(C_k) \prod_{l=1}^{L} P(F_l|C_k)$$

If desired, it is therefore also possible to specify the probability of each class separately.

3.2.2 Training

As described in the section above, a single Fern has to learn the conditional probability distributions for each of the given classes during the training phase, given a certain feature vector $\mathbf{f} = (f_1, f_2, \cdots, f_N)$ of size N.

In the original implementation, every Fern randomly selects S features and does a binary test on them, resulting in a code that can be read as a binary number between 0 and 2^S-1. When comparing this with Trees, each of these tests can be seen as the test that is applied on the previously described 'horizontally' shrunk Tree layer - instead of going left or right in a Tree node, a Fern evaluates all S binary tests leading to the required binary code. However, the original implementation was working on image patches to detect objects and the binary tests were picked by comparing two random pixel intensities in the given patch. For our initial implementation, as Kursa [2] suggests, we chose a random threshold within the range of the current feature and compare this threshold with the respective value of each sample. If the value of a sample is smaller or equal to the selected threshold, the binary test returns true and false otherwise.

Like this, we get a decimal number for every training sample by iteratively applying the random tests - this number is then used to put the current sample into one of 2^S *buckets*, comparable to hashing.
Using this pattern of dropping samples into buckets for every sample of a class that is present in the training set delivers a multinomial distribution (*histogram*) of the new features (binary tests on random thresholds) for the regarded class. These histograms are then normalized to become an actual probability distribution.
Using this method, as for many other Machine Learning algorithms, we gain information about the internal distribution with every training sample. Nevertheless, the ratio of training and testing samples should not be neglected.

To train a Ferest, we simply train L Ferns and store the specific histograms together with the randomly picked thresholds for the tests.

3.2.3 Testing

The classification of a new sample using a single Fern is very efficient: The previously stored tests are applied to the tested sample in the same order to find a binary number representing the index of the bucket into which this sample drops. The combined buckets of all classes, after normalizing, then represent the corresponding class posterior distribution given the features of the tested sample. This means that every Fern knows with which certainty a sample would be of which class.

For a Ferest, the results of all Ferns are combined using the Naive Bayes' assumption - which is now more appropriate as all Ferns check on random features and thresholds. So instead of taking the best vote as a Forest does, a Ferest multiplies all Fern probabilities to find the class that is most likely - as a result, Ferns that were not sure about their classification can be outvoted more easily as each Fern's certainty is incorporated.
If the probability of a class in the given set is available, this can be included as well - while training, this probability can be set as the ratio of samples from a certain class over all samples.

3.2.4 Adjustments

Trying to achieve more accurate results, we implemented various versions for the training function of a Fern. In this section, we will explain some of these adjustments.

We found that even for as large numbers of training samples as available in the chosen PUC-Rio data set, it is possible that some of the buckets in a Fern's histograms stay empty. Like this, a single Fern is very selective, resulting in the complete refusal of a new sample possibly being in such an empty bucket. Combining such Ferns to a Ferest results in the rejection of a testing sample when only one of the Ferns decides that the sample is not of this class, even if all other Ferns decide otherwise - this propagates through the multiplication using the Naive Bayes' assumption.
To avoid this, as suggested in [1], we assigned a very low base probability to all buckets in the following fashion:

$$p(F_z|C_k) = \frac{n_{z,k}+1}{\sum_{z=0}^{2^S-1}(n_{z,k}+1)}$$

Where $n_{z,k}$ is the number of samples in bucket z after working through the training set for class C_k. To avoid the overall undesired selective behaviour of the Ferest, this adjustment was made to all versions of our implementations.

Some approaches, that were proposed in [1] suggest to restrict the randomness in the choice of the feature and threshold for the random tests while training. Like this, we move the Fern further away from its key property of randomness for speed and more into *heuristics*.

One method that is known from the lecture on Random Forests is the random selection of feature subsets when training a single Fern. Instead of randomly choosing from any dimension, each Fern can hereby only select from the previously determined subset of features. Another common method is *bagging*: As indicated above, every Fern in a Ferest then has different training sets that allow it to be specialised on specific samples.
In addition to bagging and feature subset selection, we designed two heuristic implementations of the training method: *meanRandom* and *bestGini*.
For the first, the only difference to the normal Random Ferest is the way the thresholds for the tests are picked: Instead of uniformly choosing the threshold between minimum and maximum values, the threshold is set to the mean value of two randomly selected samples from the used training set.
BestGini includes already a lot of heuristics and actually behaves very similar to a Tree that is using Gini index as impurity measure. Instead of randomly picking the threshold, this method finds the threshold which gives the 'best' result in a sense of good and pure classification when testing on this feature.

4 Results

This section evaluates the numerous experiments that were executed within this project. Benchmarks as well as visualized results are shown for both Forests and Ferests - finally, a comparison between both is given.

4.1 Forest

As mentioned above, we were faced with some limitations regarding the choice of parameter settings used for the evaluation of Forests. Therefore, we decided two split the analysis into the following two parts:

4.1.1 Broader variety of parameters

Figure 1 and Figure 2 show our results for a broader variety of parameter settings. This induced the restriction that we only were able to evaluate these several configurations on 10% of the training data. Nevertheless, the testing was done using the whole testing set. The Trees of the Forests were trained with the stopping criteria *purity* and a threshold of 2 for the minimum number of samples in a node. The hyperparameter d was equal to 5 and the number of Trees of a Forest was in the range of $[1, 20]$.
The accuracy of the Forests was evaluated for the depths 2 ,4, 6, 8, 10, 12 and 15. On the one hand, one can see in Figure 1 that for an increasing number of Trees in a Forest its accuracy increases in the beginning but then starts to stagnate. On the other hand, the more interesting observation is that with a depth of 6 a maximum accuracy is achieved and larger depths do not lead to better results event though with a configuration of depth 15 and 20 Trees a little tendency of increase is noticeable. Being aware of the fact that this cut of increase of accuracy should be somewhere else for the complete training set, we suppose that this indicates some kind of overfitting since the depth of the Trees reaches a questionable size in relation to the relatively small training data set.
Figure 2 shows the time needed for evaluating the 55211 testing data samples. The expected linearity between the number of Trees in a Forest and the time for evaluation is totally reflected in the diagram. Moreover, our results reflect the increase of testing time among growing depths of the Trees of a Forest.

4.1.2 Training on the whole training data set

For the second part of the analysis of the Forests we fixed the depth of the Forest's Trees to 6, only built one Forest with 18 Trees and trained it on the whole training data set. The mean training time for one Tree of the Forest is 6479 seconds (1h 48m). The accuracy of this Forest on the whole testing data is depicted in Figure 3. As described above, we suppose that there is a dependence between the size of the training data set and the 'optimal' depth of the Trees. Thus, we evaluated the 100% of the training data set on a Forest with Trees of depth 6 since depth 6 performed best for 10% of the training data. As shown in Figure 3 the result is quite worse, leading us to the assumption that for the whole training data this 'optimal' depth has changed. Nevertheless, it was not feasible to evaluate other depths.

4.2 Ferest

4.2.1 Random Ferest

First, we tested the original implementation of Ferests which is very fast to train and still very accurate even for its random tests on random features:

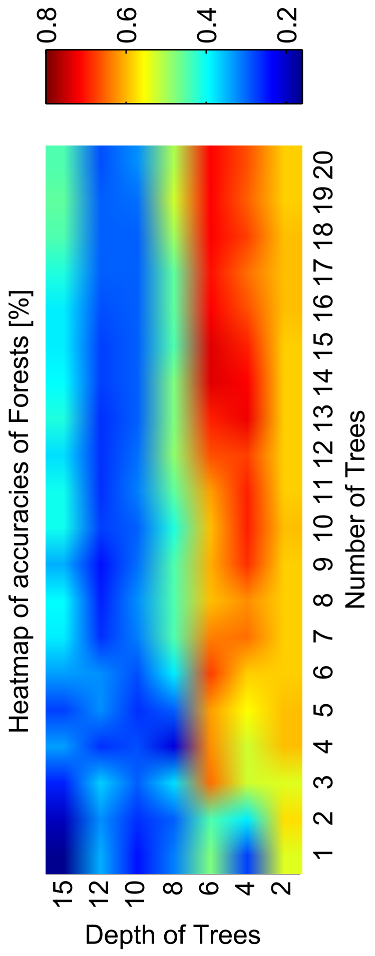

Figure 1: Accuracies of a Forest depending on its number of Trees and the depth of its Trees. The Forests were trained on 10% of the training data.

It is recognizable that the accuracy of a Ferest tends to increase with increasing number of Ferns and their depths (see Figure 4). In Figure 5 one can see that there is a limit for the best accuracy a Ferest with Ferns of a certain depth can reach - even for up to a hundred Ferns, the best result that could be achieved on the given setup for our data set was around 88%. This means that a Ferest stops improving once it reaches its maximum accuracy but rather than giving worse results it reaches a plateau and stays there. The largest improvements for this implementation were found within the range of the first ten Ferns in a Ferest.
Also, in Figure 5 the influence of chance can be seen when looking at the plots for depths one and two. Due to the choice of a random threshold, it is possible to choose a bad value or feature. This leads to the depicted

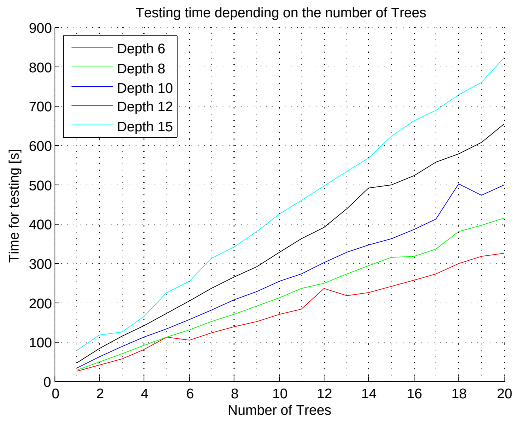

Figure 2: Testing time of a Forest on the whole testing data set depending on its number of Trees. The data was evaluated for five different depths of the Trees of a Forest.

unsteady behaviour - especially for Ferests with low depths this effect is quite strong. The same characteristic can be seen in Figure 4, e.g. for depth 6 and 25 Ferns in the structure. This effect seems to dissolve the deeper a Fern becomes.

Nevertheless, the depth of a Fern can also become critical when chosen too large. Where Trees tend to overfit for such settings, the number of buckets for the histograms of a Fern increases dramatically. Accordingly, very few samples drop into each bucket while training and therefore, the class-conditional probability distribution is close to being uniform. Informally, every Fern is then uncertain to which class a new sample might belong.
This shows that when increasing the depth of a Ferest, the number of samples in the training set should increase exponentially - for our chosen training set, the best depth seems to be ten.

Due to restrictions in time and computation power, we could neither further increase the depth of the used Ferns nor the number of Ferns. But for single samples the following assumptions were confirmed:

- After reaching a certain depth, a tendency of decreasing accuracies occurs
- For depths that are higher than a certain threshold, classification accuracy steadily increases for larger

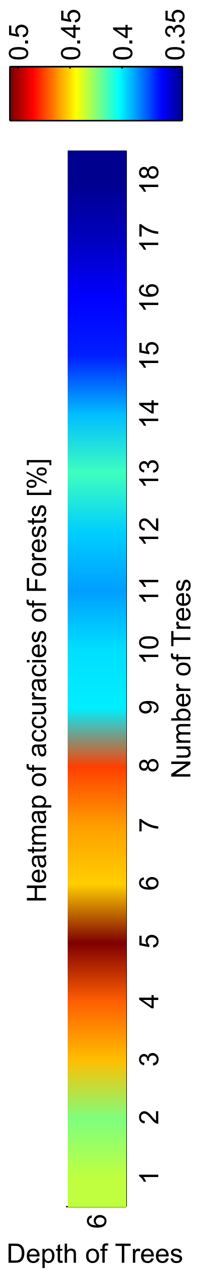

Figure 3: Accuracies of a Forest depending on its number of Trees and the depth of its Trees. The Forest was trained on 100% of the training data.

Fern-numbers until it stagnates at a certain value

4.2.2 Optimized Implementations

Additionally to the original 'completely' random version of the Ferest, we implemented *bestGini* and *meanRandom*. But where training of a single Fern of the same depth

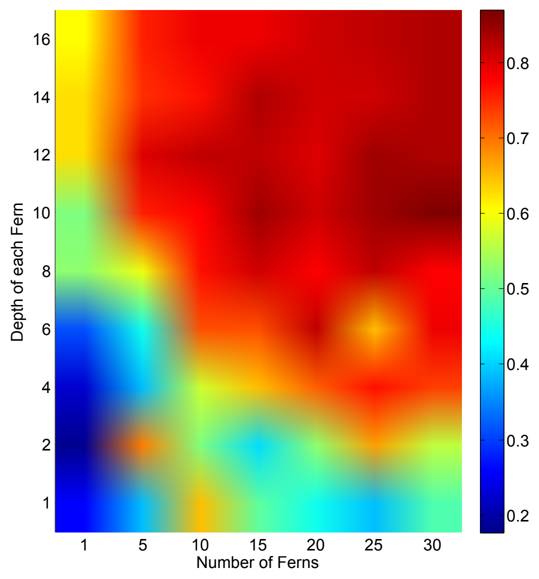

Figure 4: Heatmap for accuracies of a Ferest with different depths and numbers of Ferns

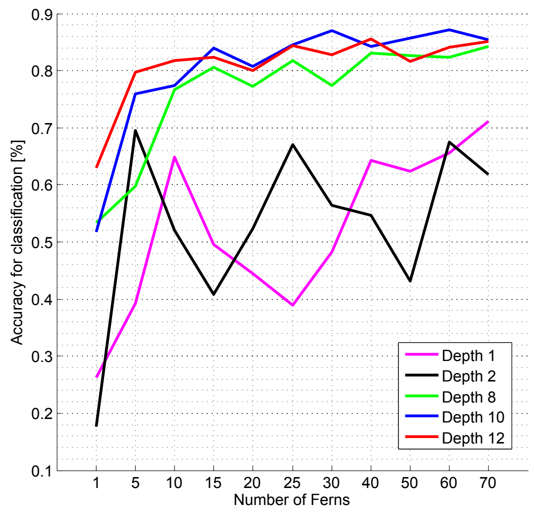

Figure 5: Accuracies of a Ferest for large numbers of Ferns

took only 15 seconds for 'random' implementations, it already lasted 20 minutes for the heuristical option incorporating costs with Gini as impurity measure. Nevertheless, *bestGini* was still faster for the training by orders of magnitude (see next section about Comparison).

When comparing the classification accuracies of the three different versions that were implemented, in line with expectations, bestGini most often gives the best result (see Table 1). One can see that for larger numbers of Ferns per Ferest, all three implementa-

D	*N*	*rnd*	*mean*	*Gini*
8	1	53.34%	66.26%	73.25%
	5	59.72%	79.82%	81.20%
	10	76.64%	86.52%	83.80%
	15	80.59%	86.34%	82.68%
	20	77.30%	85.65%	86.93%
10	1	51.68%	60.66%	62.05%
	5	75.96%	77.09%	78.69%
	10	77.41%	86.63%	81.97%
12	1	63.03%	41.31%	53.33%
	5	79.66%	74.68%	76.19%
	10	81.81%	88.38%	81.42%

Table 1: Accuracies of a Ferest with depth D, number of Ferns N for original *random*, *meanRandom* and *bestGini* implementations

tions deliver comparable outcomes, whereas bestGini generally gives the best initial result for single Ferns. Surprisingly, meanRandom achieves much better accuracies than the original random version. We suppose the following explanation for this phenomenon:
Firstly, taking the mean of two random values for certain feature dimensions is still a random choice, so it is comprehensible that *meanRandom* should not achieve worse accuracies than pure random. The reason for the advance probably is of statistical origin - when having a fixed feature dimension with minimum and maximum value there must be some areas where samples of a certain class accumulate. If this was not the case for all dimensions, it would generally be impossible to successfully learn appropriate distributions. Assuming we need to find a threshold for a feature where the class distribution resembles the sample distribution (meaning that there are different areas that are common for samples of a specific class). When we pick only one random value from those samples, the chance to pick a sample that is in the middle of such a 'crowded' class-area is comparably high - resulting in a split that separates the class evenly (this is not desired). If we pick two values instead, the chance of taking two samples from different class-specific regions is comparably high. Like this, we split this feature in the middle between the two regions and thereby, most probably, also split the two related classes. Therefore, if we combine enough Ferns on hopefully distinctive features, the output will be a very good classification.

D	*Tree train*	*Fern train*	*Tree test*	*Fern test*
8	279.89	17.05	220.42	12.31
10	350.49	16.17	266.29	13.62
12	351.62	19.80	349.26	15.17

Table 2: Comparison of average training and testing times on the whole data set for different depths D of Forest and Ferest

4.3 Comparison

The two sections above concentrate on the behaviour of the two main Machine Learning algorithms that were implemented. The following last section gives a brief overview on differences between Forests and Ferest as well as some comparisons regarding computation time for training and testing for each of the structures.

As it was not feasible to test all possible settings of the Forest for a best accuracy within this project, it is hard to compare classification accuracies between Forest and Ferest. Theoretically, when using the best suited impurity measure, Forests should be at least as accurate as random Ferns. Especially, when comparing the *bestGini* implementation with Forests using a Gini index, behaviour and results should be almost exchangeable. However, as we only have subsets of experiments for all structures on the same data we will not further describe the differences in classification accuracy.

Instead, we want to point out the computational speed of Random Ferests. Table 2 shows the immense advance of Ferns over Trees in terms of times necessary for training as well as testing. We compared times for different depths and found that Ferns are 15-25 times faster than Trees for both phases, even for the Forest being trained on only 10% of the data. Incorporating single experiments on the whole data set, Ferests train up to 50 times faster than Forests under the same conditions.
The speed in training is clearly due to the quick and uncomplicated selection of a random threshold for the binary tests. Searching the best feature and value for a split at a specific Tree node by comparing costs using a Gini index is comparably very time-consuming.
The significant differences in testing times are unexplainable to a certain extent. Given the structure of both implementations, evaluation works very similar - S tests are ap-

plied to the given sample and thereafter the maximum probability is chosen. We assume the reasons for the difference lie in the efficient implementation of Ferns when applying all tests at once instead of 'going through' the Trees to select the appropriate tests - a good compiler might parallelize this section for Ferns but not for Trees as the result of the previous test decides which test is applied in the current node. Additionally, we assume it might partially be due to the chosen programming language (Matlab - "MATrix LABoratory") as for Trees, we're not only using matrix operations but also many other structures (object-oriented) for a good readability and re-usability. Maybe those testing times would be different when using Python or C.

Lastly, we can say that Ferns can very well be applied to data with categorical and continuous values without changing their behaviour. Especially results for the adjusted *meanRandom* implementation of Ferests are very convincing and should be further investigated.

References

[1] M. Özuysal, P. Fua, and V. Lepetit. Fast Keypoint Recognition in Ten Lines of Code. *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1-8, June 2007

[2] M. B. Kursa. Random ferns method implementation for the general-purpose machine learning. *Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw*, February 2012