The random forest kernel

Martin Hallen
School Of Computer And Communication Sciences (IC)
École polytechnique fédérale de Lausanne (EPFL)
Lausanne, Switzerland
Email: martin.hallen@epfl.ch

Prakhar Agrawal
School Of Basic Sciences (FSB)
École polytechnique fédérale de Lausanne (EPFL)
Lausanne, Switzerland
Email: prakhar.agrawal@epfl.ch

Luc Zeng
School of Engineering (STI)
École polytechnique fédérale de Lausanne (EPFL)
Lausanne, Switzerland
Email: luc.zeng@epfl.ch

Abstract—**In this report we look at the Random Forest Kernel introduced by [Davies and Ghahramani(2014)]. We describe the methods involved to construct such a kernel from methods that is not normally used for random partition. The kernel is built by selecting random partitions from a Random Forest created by the training data. This is an instance of a supervised kernel.**

We first go through the theory of Random Forests before we describe how it is possible to sample from this. Then we introduce the kernel and it's construction before we show examples of regression results and compare the kernel with other known kernels.

I. Introduction

Kernels are a very important tool of machine learning algorithms. Indeed, they allow to implicitly map the features onto a better suited space for discriminating between points. However, the new space is usually higher dimensional, making the problem computationally difficult. The **kernel trick** makes the inner product between two vectors in the new space computationally feasible.

The commonly used kernels are usually **unsupervised**. Though having proven their worth, they usually don't adapt to the underlying statistics of the data. We have a wide range of known kernel methods, as the Linear kernel, Periodic kernel, Radial Basis function (RBF) and Polynomial to mention some of them. The most popular kernel is the RBF kernel, and it is often presented as the default kernel.

In this report, we describe a supervised kernel method called the Random Forest Kernel. We show how we can sample random partitions from the dataset using an implementation of the Random Forest.

The implementation of this algorithm is Python. We use the Python library Scikit-Learn for the implementation of the Random Forests and SVR.

In the project we have:

- Successfully implemented the Random Forest Kernel. To that end, we also wrote a function(with the help of some sites) using which we can traverse and get almost every kind of information about the tree(most users just want the output of the forest).
- Understood the underlying process of the random forest algorithm and explained the behavior of our kernel
- Carried a series of test of the kernel for the regression case. We have compared it to RBF kernel.

II. Methods

A random forest is an ensemble of decision trees. We therefore want to describe the structure of the decision tree first to simplify our finding later in the report.

A. Decision Trees

This section and the next mainly rely on the knowledge given in [Tan et al.(2005)Tan, Steinbach, and Kumar].

Decision Trees are tree-shaped object used for prediction. They are made of :

- **Root Node** It is the first node of the tree. All the data points will go through it. It separates the data in two parts given the a value of one of their features.
- **Internal Nodes** They each receive data points and lead to new nodes. As with the root node, they perform tests to separate the incoming data in two parts.
- **Leaf Nodes** They are the end nodes of the tree. They all represent a class, or a subdivision of the space.

In figure 1 we give an example of a tree on a toy dataset. The data set is: $X = [1, 2, 3, 4, 5]^T$ and $y = [10, 5, 1, 5, 10]$.

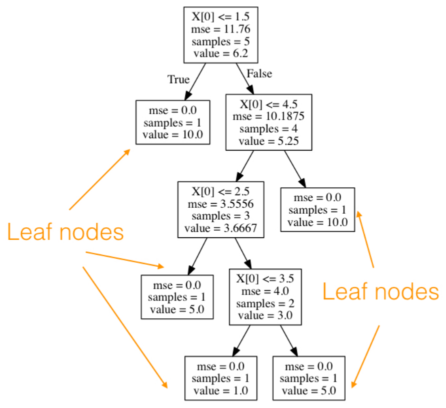

Fig. 1. Decision Tree

B. Decision Tree learning

Though we are using already built decision tree learning algorithms, we give some insight on how the trees are learned. It will be of help later when we describe the behavior of the algorithm. The goal of this section is to only give a basic insight of decision trees.

The basic idea is to select a feature and a threshold, grow the tree and stop when all the points agree on the label. The Hunt algorithm is the idea on which tree learning algorithms are bases.

Consider a node D. The algorithm is as follow:

Algorithm 1 Hunt algorithm

```
1: If all the points in D have the same label, D is a **leaf node**
2: Otherwise, an **attribute test condition is created** that separates D in two different nodes. The points in D are then distributed in the **children nodes of D**
3: Stop when there are only leaf nodes left
```

The main question is now: **How do we choose the attribute test condition?** We will briefly explain this in the 1D case. In order to split the data at one node, the main idea is to **increase the purity of the points**. There are many metrics that estimate the purity of a node. In this project we have used the one that follows the Scikit-Learn distribution. Scikit uses the MSE as the split function. Calling the labels at node D y_i, N the number of points present at this node, the MSE of node D is :

$$MSE(D) = \frac{1}{N}\sum(y_i - c)^2$$
$$\text{with} \quad c = \frac{1}{N}\sum_i y_i \tag{1}$$

This quantity basically evaluates the spread between the points at node D. Now, to select the best threshold, we define a criterion called the *gain* that evaluates the difference in purity between the current node and the proposed children:

$$\Delta = I(parent) - \frac{N_1}{N}I(D_1) - \frac{N_2}{N}I(D_2) \tag{2}$$

where D1 and D2 are the children nodes, I is the purity metric (MSE for regression).

The decision tree algorithm is then to find the split that maximizes Δ. The algorithm that Scikit-Learn uses is called "CART" and is based on the idea presented here.

C. Random Forests

A Random Forest is an ensemble of decision trees. We call the decision trees for estimators. Random forests follow the same pattern as bagging, except for one step. In addition to selection a subset of the points to create the estimator, the estimator is also built **using only a random subset of the available features**. Indeed, for the variance to diminish, each tree should be the most uncorrelated from one another as possible. As some points and features will have more importance than other for the prediction, they would be more presents, leading to correlated trees. The random forest algorithm:

Algorithm 2 Random Forest constuction algorithm

```
**for** i=1 to m **do**
    Select a random subset (with replacement) A$_i$ of the data points $X$ with size(A$_i$) $\prec$ size(X) and a subset of the features. Denote this subset as $F_i$.
    Build a decision tree T$_i$ from A$_i$ using the feature set $F_i$
**end for**
Average the T$_i$ : $f(x) = \sum_{i=1}^{L} T_i(x)$
```

where m is the number of estimators.

D. Bagging

Bootstrap Aggregating, or more commonly known as Bagging, is a common Machine Learning technique that is designed to increase the accuracy of simple predictors. It is shown that it reduces the bias while having little effects on the mean. Bagging consists in randomly selecting subsets (with repetition) of our dataset, constructing the subsequent tree and finally averaging (voting in the case of classification) the forest. The algorithm is thus:

Algorithm 3 Bagging algorithm

```
**for** i=1 to L **do**
    Select a random subset (with replacement) A$_i$ of the data points X with size(A$_i$) $\prec$ size(X)
    Build a tree T$_i$ from A$_i$
**end for**
Average the T$_i$ : $f(x) = \sum_{i=1}^{L} T_i(x)$
```

E. The Random Forest Kernel

The algorithm that we will describe aims to create a **partition distribution**. Partition distributions are interesting because they allow uncertainty in the classes of the data.

A partition distributions P represents how likely each cluster of the data is. The question is: how can we build P? As it is quite tedious to define an analytic PDF for partitioning, we'll use another idea: a partition distribution P is any program that randomly defines a partition of the data.

This is where random forests come in handy: being inherently random and outputing a partition of the data, they **define a partition distribution**. **The question is then: how do we define the kernel?**

[Davies and Ghahramani(2014)] prove the following: given two data points a and b, a kernel can be defined as the number of times on average that the kernel assign a and b to the same cluster. In short:

$$k_P(a,b) = \mathbb{E}[I(\rho(a)=\rho(b))]_{\rho\sim P} \tag{3}$$

where I is the indicator function.

They show that by using a finite number of samples from P, we obtain a good approximation of the kernel. The kernel is thus approximated by:

$$k_P(a,b) \approx \frac{1}{m}\sum_{\rho\sim P}^{m} k_\rho(a,b) \tag{4}$$

where m is the number of estimators. In this notation, $k_\rho(a,b)$ is 1 if ρ assigns a and b to the same cluster, 0 otherwise.

F. Algorithm

We have yet to define the partition distribution. We make use of the following observation: a tree generated by the random forest algorithm constitutes a **hierarchical partition of the dataset**. In other words: **by randomly selecting any height of one tree, we define a random partition**. The partition is thus done by taking the ancestors of the leaf nodes. If two data points have the same ancestor, they belong to the same cluster.

Algorithm 4 Random Forest Partitioning Sampling Algorithm

```
Input: $X \in R^{NxD}, \vec{y}^t \in R^N, h$
$T \sim$ RandomForestTree$(X, vecy^t)$
$d \sim$ DiscreteUniform$(0, h)$
for $a \in D$ do
    leafnode $= T(a)$
    $\rho(a) =$ ancestor$(d,$ leafnode$)$
end for
```

In figure 3 we show an example of our algorithm on the same data as before. In this case, the selected height is 2. We can see that the 3 bottom points have the same ancestor: the kernel will be the same for these points. The point which value is 10 at height 2 won't move. We observe however that the point of value 10 at height 2 doesn't have any ancestor. This is, as we will extensively explain, a major issue of this kernel. Indeed, **the points that are very close to the root can't be classified**.

[Davies and Ghahramani(2014)] simplifies in their paper the selection of the height h. Their explanation assumes that we have a balanced binary tree of 2^h nodes. In practice this is not true, and we can show that this is important for the algorithm to give the correct precision. Figure 2 displays the result of doing this.

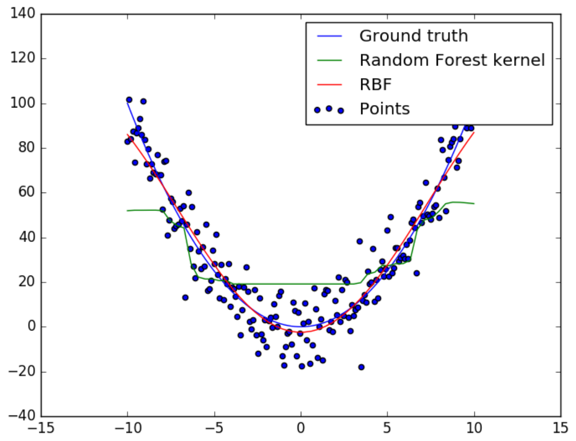

Fig. 2. Choosing the minimum height of the tree causes the function to flat in certain parts.

If we define the height of the tree to be the minimum height of a leaf node, then we fail to distribute the points deeper in the tree.

An alternative to this approach is to define the height h of the tree to be the maximum height of the tree. We then have to define what happens with nodes that are more shallow in the model. Figure 3 displays the problem.

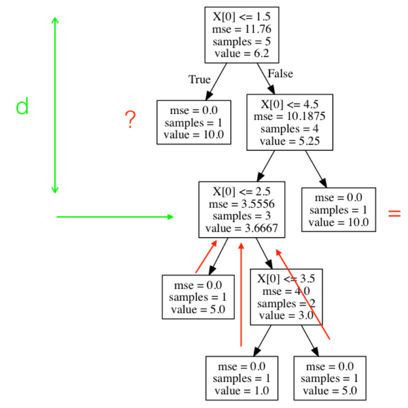

Fig. 3. Selecting the correct height in an unbalanced decision tree. A challenge is to define what happens with shallow leaf nodes.

If a node a is shallower in the tree than the picked height d, we define the ancestor as the node itself. More specifically, $\rho(a) = \text{ancestor}(d, \text{leafnode}) = \text{leafnode}$.

In practice, this means that the partitions of the datasets will include partitions of datapoints placed in the same leafnode above the height d in the tree.

The reason behind the flat part of the curve in the middle of figure 3, is that the algorithm for the decision tree tries to minimize the MSE in the split of a node. For the square function, this means that the points in the end of the graph will be separated out first, and therefor lay more shallow in the tree.

So basically our algorithm doesnot have any parameter other than m(number of partial kernels). Larger value of m gives more accurate results. Only one parameters allows great advantage of Random Forest Kernel over RBF as can be seen later in the report. Our algorithm tend to overfit if dataset is very low dimensional because of the reduced randomness of the forest in the algorithm. It was shown by [Davies and Ghahramani(2014)] that Random Forest kernel seems to scale as approximately O($N^{1.5}$), while the RBF remains faithful to its theoretical O(N^3).

G. *Implementation with Scikit-Learn and SVR*

When working with a pre-computed kernel as we do with the Random Forest Kernel, there are some differences in the implementation. The SVR can not calculate the kernel distance between two points it has never seen before. The solution to this is to create a kernel that includes both the training points and the testing points. Note that this does not mean that the testing points are being used during training.

The training points are fed to the Random Forest algorithm and are the basis of constructing the ensemble of decision trees. This is the training part of the algorithm.

We then construct the kernel matrix between all the data points, both the training points and the testing points. The kernel therefore define the distance between all data points, as is required by the SVR.

When we train the SVR, we use only a subset of the kernel, namely only the internal distance between the training points. This has the same effect as training the SVR on the training features.

Testing the SVR is done by providing the kernel matrix defining the distance between all pairs of training and testing point.

A graphical representation of the kernel can be seen in figure 4. This kernel is created on a toy data set for the square function $f(x) = x^2$. We have sorted the training points and the testing points by the x-value and made sure all training points come before all testing points.

We can see a clear square in the upper left corner. Since our training features is equally separated x-values, we can see that the distance between point is the middle of the kernel are shorter. In the center of this square we see the points that with a x-value close to zero. The derivative of the $f(x)$ is small in this area, and the point have a higher change of ending in the same partition during the construction of the Random Forest kernel.

The lower left part of the kernel shows the kernel computed between the training points and the testing points. This is effectively what the SVR uses during prediction of testing points.

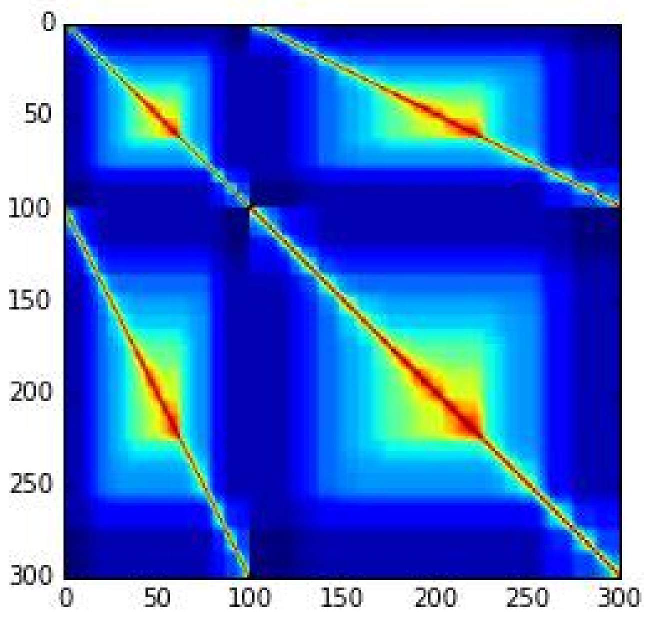

Fig. 4. Example kernel over the square function $f(x) = x^2$

III. Results

All the results in this section are obtained using SVR with the Random Forest Kernel. We first show results on 1D datasets and then move on to more complicated datasets.

A. 1D dataset

We below show some results from the sinus function and the square function.

We observe that the performances are much worse for the square function than the sinus function. Our theory is that this is due to the **imbalance of the data**. Indeed, for the sinus, there are roughly as many instances of points in each range. For the square function, the density decreases as we move away from the origin. **Imbalanced data leads to imbalanced tree. This means that some values will be very close to the root node, because the tree construction algorithm imposes the highest purity possible in the new nodes (see Part II).** In the first example of this report, we can see that the values "10" are separated first. This is the general behavior of decision tree-building algorithms. As a conclusion to this observation, **we can state that the Random Forest Kernel is most suited for balanced datasets**.

Another way to understand this (see [Breiman(2001)]) is that the algorithm tries to minimize the overall error rate. It will thus keep the error low on the dominant class, and let it be high on the smaller classes.

There are some fixes that can be done for imbalanced type datasets: sample less in very common classes or apply a weight to each point that compensates for the imbalance. The robustness of random forest to unbalanced data is a well known issue and is still a research topic in the community.

B. Multidimensional datasets

We compare the results of using the Random Forest kernel against the RBF kernel using SVR on 5 real-world regression problems from the UCI Machine Learning repository. Random Forest Kernel was trained with $m = 400$. The comparison is done using Normalized MSE. The hyper-parameters of the RBF-kernel are done though grid search.

We noticed from figure 5 that Random Forest Kernel performed better then RBF kernel in most of the cases. Note that this might be due to a bad selection of hyper-parameters in the RBF kernel. Selection of hyper-parameters is difficult and require some knowledge about the dataset. One of the main advantages with the Random Forest Kernel is the lack of hyper-parameters. The algorithm learn by itself how the datapoints are best separated in feature space.

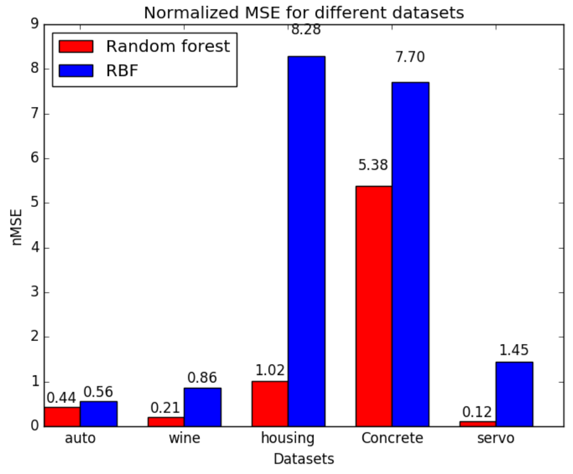

Fig. 5. The normalized MSE for 5 different real world datasets. The Random Forest outperforms the RBF kernel in all cases.

To understand the effect of the kernel approximation on the resulting predictions, we plot the test MSE performance of Random Forest kernel for different values of m. In figure 6 we can see that for low value of m as well, the Random Forest kernel performs nearly as well as its optimum.

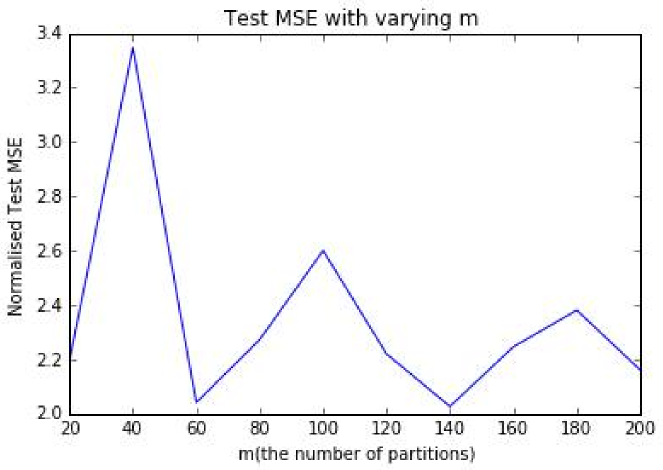

Fig. 6. We can see that the means square errors decreases with the number of partial kernels m used to construct the kernel. This image only shows a trend, but we expect the result to be confirmed by a deeper analysis.

We also tested Random Forest kernel by taking different number of features of "Housing" dataset. We comparedthe normalized MSE with features and we found that Random Forest Kernel performed better with more features. This is quite evident as Random forest Algorithm partitions the space better with more features.

To give a graphic representation of how the Random Forest Kernel predicts values compared to the RBF kernel, we see the difference between the predicted value and the actual value in the wine quality dataset in figures 7 and figure 8.

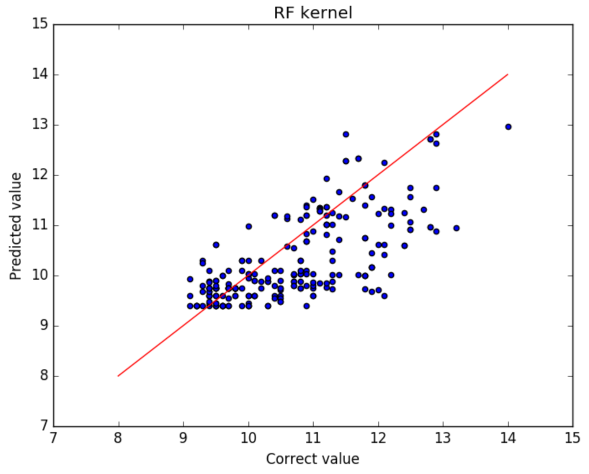

Fig. 7. Actual and predicted alcohol concentration on the wine quality dataset for the Random Forest Kernel using SVR. The red line displays the function $x = y$

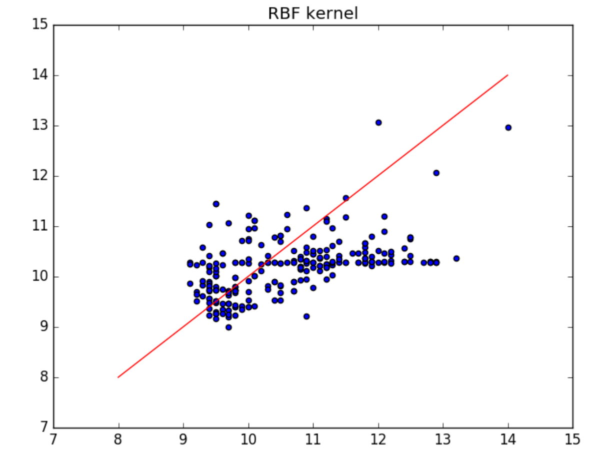

Fig. 8. Actual and predicted alcohol concentration on the wine quality dataset for the RBF Kernel using SVR. The red line displays the function $x = y$

C. Pros and Cons of the Random Forest kernel

The kernel has a number of advantages over classical kernels:

- It has no hyper-parameters. No expensive selection of parameters is needed as in other methods. This is often a problem with other kernels, as underlying information about the data might be needed
- It supervised.

These advantages come at the cost of :

- The training is slow. However the code is obviously parallelizable.
- It is somewhat dependent of the type of data we use. But there is hope that this can be fixed.

IV. Conclusion

We have implemented Random Forest Kernel, a novel method for constructing effective kernels based on a connection between kernels and random partitions. We can conclude from different observations that our kernel performs well but is also dependent on data. It works well on high dimensional datasets and improves with number of partitions. Our kernel can still be improved by implementing an efficient balancing scheme.

References

[Breiman(2001)] Leo Breiman. Random forests. *Machine learning*, 45(1): 5–32, 2001.

[Davies and Ghahramani(2014)] Alex Davies and Zoubin Ghahramani. The random forest kernel and other kernels for big data from random partitions. *arXiv preprint arXiv:1402.4293*, 2014.

[Tan et al.(2005)Tan, Steinbach, and Kumar] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining, (First Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2005. ISBN 0321321367.