Comparison of Tree based Learners in Incremental Dataset of Software Defect Predictions - A Case Study

Md Rayhanur Rahman*
Ph.D. Student
Dept. of CSC
NC State University
Raleigh, NC, USA
mrahman@ncsu.edu

Abstract

In various research fields, data miners are being applied in an intense manner such as in domains of computing, space, business intelligence etc. In software engineering domain, it is also being used extensively. One of the key area where it is being applied is defect prediction. Defect prediction models helps software developers to control the quality issues of the software projects. There are a diversified range of data miners utilized to predict defects from the software metrics such as simple regressors and classifiers as well as complex multi objective models. These days, source codes of the software can be obtained from the github and other repositories easily and code metrics can be computed on the fly. This indicates that datasets for defect prediction can grow to a large volume incrementally over time. In such scenario, scalability challange will appear as the data become larger and larger. In this research, we will try to compare the offline tree based classifiers and online VFDT classifier to observe the case of classifiying defects from large datasets. From our observation, we found out that, online classifier behaves more stable and produces similar results without paying the penalty of time.

Keywords Defect Prediction, Decision Tree, Random Forest, VFDT, FFT, Online Analysis

1 Introduction

These days, software automation has engulfed all the spheres of our life. Millions of software companies automates new business logics along with replacing legacy systems with its modern descendant. Consequently, software development is an ongoing process that never stops and hence, there will be always defects in the software sources that needs to be fixed in constant manner. Fixing these defects is one of the key concern in software quality assurance activities and dedicated human resources spend a significant time in resolving these issues - if unfixed, culminates in loss of money, time, consumer satisfaction and in mission critical cases, casualties [6].

Finding and fixing software defects was a manual task decades ago. But these days, data miners are there to help the developers find these defects. These data miners work on the defect prediction models that mainly contains software code quality metrics such as line of code, cyclometric complexity etc. Based on these data, those data miners predicts whether a particular software module would contain defects or not.

...To perform defect prediction studies [7, 14], researchers have explored the use of various classification techniques to train defect prediction models. For example, in early studies, researchers used simple techniques like logistic regression and linear regression to train defect prediction models [1, 8]. In more recent work, researchers have used more advanced techniques like adaptive regressions, ensemble learning etc [2, 4]. Several context sensitive analysis based defect predictors are explored as well [11].

...Despite the fact that, there have been numerous advanced miners deployed in defect prediction; classification techniques to build defect prediction models have focused on the performance. However, as the volume of software source codes increases in daily basis, so does the size of defect prediction examples from which the classifiers will predict the defects. Hence, in case of traditional learners, memory and sample size will be a dominant obstacle if those are fed with incrementally large amount of data from time to time. Being trained with small number of data also suffers from overfitting. Meanwhile, currently there are many online data-miners are available which are sample size agnostic. Thus exploring the comparison of traditional learners and online data miners have become a priority.

In this paper, we have put three offline classifiers named CART decision tree based classifier, Fast Frugal Tree (FFT), Random forest against online classifier named VFDT [5]. We will look into three research questions in particular:

- **RQ1:** How these four classifiers performs in large defect prediction datasets that will increment over time
- **RQ2:** How hyper-parameters changes as the data size changes

*unity id: 200255928

NCSU, FALL 2018, December 7, 2018, Raleigh, NC, USA
2018.

- **RQ3:** How much computation resources would be used by these four learners

We have used four defect prediction datasets obtained from (...). These datasets contains around 40, 000 examples on an average. From our observation we found out that, VFDT performs better and more stable manner than the other three as the data size increases. Moreover, the performance differential becomes more prominent among those learners considering data size increase and finding tuning parameters.

Rest of the paper is organized as follows. In the section II, we will discuss existing literature study on this aspect. In section III, we will discuss baseline criteria upon which we will evaluate our comparison of the learners, In section IV, we will discuss the experiment setup and result analysis. Finally, it will be followed by discussion and future work scope in the section V.

2 Related Work

3 Background and Motivation

...

3.1 Baseline Criteria for Data Miners

These days, numerous data miners are being deployed in SE domain. There are also a diversified variation in how those miners tweaked and tuned to fit a particular model. That leaves us with virtually unlimited choice to pick a certain data miner or optimizer to apply in a new problem. However, according to the no free lunch theory, these is no single miner which would suit best in every possible models. So, we need to apply commissioning to filter through the possible choices of data miners or optimizers. In that regard, baseline criteria is truly useful. Baseline criteria refers to the important factors, majority of which should be achieved by a learner. It provides key insight to us regarding how effective and efficient the data miners would perform in the real world. Here follows some of the key baseline crietria along with their short description.

3.1.1 Simple and Reasonable

Learners should be simple in a sense that it is easily explicable to the end users. It should also be easy to understand the underlying models, how it works and how to work on to build on further. There are several learners that not very easy to describe such as Naive Bayes classifiers and neural networks. On the other hands, decision tree based models are simple enough to understand and explain to others. However, learners have to be reasonable as well. It should perform well in terms of accuracy and performance. Otherwise being a simple learning producing non-reasonable results does not help the case of data mining activities.

3.1.2 Stable and Robust

Data miners work on examples to build the model and on the basis of that model, make the predictions. Datasets containing loads of examples poses a significant challenge for the miners. Datasets comes with lots of unique cases as well context aware information. Some datasets are also imbalanced while the others might be incomplete. There is also a potential chance of a lot of anomalies hidden in the dataset. All these factors contribute to the fact of being a data miner unstable. This means the learner would produce different type of decisions in case of diversified datasets which is extremely frustrating for business users. It also prevents further improvement of the data miners. Data miners also need to be robust throughout most of the cases of different sample size, splits and validation techniques.

3.1.3 Generic

Data miners should be as generic as possible referring to the fact that they should provide a range of possible solution rather than a simple one point one. The benefit of it is to help the end users with a possible wide outlook of the scenario. But if the miners behave too specific or provides only one solution of a certain scenario, it would confuse the end users more. There is also a chance to miss other corner cases and ignore other possible, equally similar solutions which would turn out truly bad in real world scenario.

3.1.4 Replicable

The learner should build model from the dataset and produce outputs fast enough. This ensures that the learner can be invoked iteratively in case of learning, understanding, rebuilding or tweaking. If it takse several hours to produce outputs for a small data, it would become impossible to work on that learner or tune a little bit. At the same time, the learner must also produce the same output every time if is fed the same input. If the outcome of a certain learner is not replicable, then it is impossible to understand, implement and improve the learner.

3.1.5 Goal Aware

These days, all the real world problems do not focus on a single goal. Earlier, optimizers were used to maximize or minimize a value over a set of constraints but now a days, complex situations are there where there is no optimal solution. Hence, data miners being applied in those fields should be goal aware. This means, rather focusing on a single goal, the miner should produce output in such a manner so that the output should reflect overall realization of multiple goals. There might be conflicting goals and often, the correlation of the goals and datasets are quite complex. None the less, the miner should be able to handle such conflicts, complexity and tradeoff cases and produce results that provides a satisfaction of all the goals.

3.1.6 Anomaly Aware

It is almost certain and usual that the datasets would contain a percentage of noisy and anomalous examples. Data miners should be able to detect and cancel out those. However, sometimes data miners confuses with the true example as anomalous examples and thus discards those or learn something misleading. A good data miner should be able to handle all these cases.

3.1.7 Context Aware

Despite the fact that a dataset should represents a generic scenario of a particular phenomenon, often this is the case that many context specific information are hidden in the datasets. So in the datasets, there might be a region of locality which is almost similar to other generic examples but might vary in one or two attributes and indicates different labels. Those regions are in fact, responsible for difference in decision making process. Thus, a good data miner should be able to find out those context specific local regions and learn the phenomenon protecting the generic learning model intact as well. This baseline is very critical in fields of health, e-commerce etc.

3.1.8 Incremental

Modern days, datasets tend to be large. They also get incremented periodically. So it is in vain if a data miner build models from a datasets and then can't reuse the already built model if it wants consider the newer example. So data miner should be able to extend the existing knowledge of model reading the newer examples. It should also be able to run over infinite stream of datasets as well as relearn in case of anomalies.

3.1.9 Shareable

The datasets and the obtained knowledge by the learner should be shareable so that the obtained learning can be obtained in different contexts. However, while sharing the datasets, privacy should be a big concern. So miners should be able to hide the actual data and interpret the outcomes at the same time so that sensitive information of the datasets are protected.

3.1.10 Tunable

There are a variety of parameters on every learners that can affect the accuracy, performance and stability of the learners. However, for a large datasets, if it is necessary to find the best set of tuned parameters for the miner, then it would take a hefty amount of time to do so. So a data miner should be able to tune itself so that it can be fit to a certain dataset or certain domain specific models to obtain better results.

In addition, there are several other baselines that deserves mention:

- Be applicable to mixed qualitative and quantitative data
- Have no parameters within the modelling process that require tuning.
- Be publicly available via a reference implementation and associated environment for execution
- Be computationally cheap in a reasonable sense

3.1.11 Criteria for this Work

All of the aforementioned criteria are more or less critical for any learner. However, set of baselines critical for a particular domain depends on many contexts as well. In some cases, computational performance might be the most critical. In other case, the accuracy might be the most critical one. In the context of this work, we will try to observe the performance of various tree based learners on incremental data. Hence, for our work, here are the list of critical baseline we will look more closely:

1. **Usefulness and Reasonable:** The output of the miners should be at least better than some random guess. In this work, the miner would predict whether a given example contains bugs or not. The typical evaluation metrics such as precision and recall of the learners would be put into the focus. The examples will be fed into the miners in an incremental manner and we would observe how those evaluation measure changes over the new datasets that fed into the learners. Also the outputs need to better than the established learners.
2. **Stable:** Stability is quite important in case of prediction. If the output of the learners produces varying degree of outputs, then the decision making process would be frustrating. In the context of this work, we would look into the stability of the prediction of the learners in terms of incrementally updated set of examples and observe how these learners perform.
3. **Cheap:** The computational resources being used by the learners is another important aspect. Learners might produce great results but if it comes using heavy computational resources such as cpu time and memory, then it might not become that useful in real world scenario. In this work, learners will learn from a range of small datasets to large one. So, we will try to gather information on how much cpu time and memory resources those learner will use to produce results.
4. **Robust:** In case of incremental datasets, the learners need to accommodate all the newer examples. However, this might also confuse learners with different examples or noisy examples. So it needs to be seen how robustly the learner can handle all the examples that will be added in the larger datasets.
5. **Streaming:** In case of very big datasets, the learners should be able to stream in the newer examples and fit

those in the existing models. Otherwise, the learners need to completely relearn everything in case of the arrival of the newer data. In the context of this work, this is a very critical requirement.

6. **Tuning:** Tuning of the parameters of the learners is another important baseline. In our case, we would feed the learners with increasingly bigger amount of example set and observe how the initial tuning performs for the rest of the data. We would also like to observe how the optimal set of tuned parameters changes when newer examples comes into the context.

3.2 Motivation

In the domain of software defect prediction, several learners have been applied which includes logistic regression, neural networks, decision tress, ensemble learning, evolutionary algorithms and context sensitive analysis [1–4, 8, 11, 12, 15]. All of these learners load the whole data into primary memory and performs the classification task. However, things have changes rapidly over years. These days, source codes are being committed to the source code repository periodically. The frequency of committing newer code base is ranged from hours and to days. In order to help developers with new potential defects while the new code base being committed, need to relearn the whole defect prediction model is necessary. As the new code is being pushed, software source code metrics which are being used be defect prediction models needs to be recomputed. Hence, the newer examples of software defects would come into play. So, assuming that newer version of source code being committed to the repository at least once in a day results into re-computation and relearning the whole things from the scratch. Traditional decision tree based learners such as Classification and Regression Tree, Random Forest, Fast frugal Trees all would need to relearn everything which is very cpu and memory extensive. If, high speed stream based learners would have been used in this scenario, then there would be no need to relearn everything from scratch as the streaming data miners automatically accommodates the newer examples with the existing models. That signifies the motivation of this work. We would like to observe how streaming learners perform against the offline learners in terms of feeding newer set of examples periodically. Hence, in this work, we would compare CART, RF and FFT against VFDT which is an online stream based decision tree learner and compare all these learners from the aspect of aforementioned baseline criteria.

3.3 Classifiers

In this work, we will observe the performance of four decision tree based classifiers. Here is a short description of each of these classifiers.

3.3.1 Decision Tree

Decision Tree learners uses decision tree based models as a predictive model to perform classification or regression over a set of examples. In our case, we will only use classification trees where the tree will be used to predict the class labels of a given set of examples. Each interior node refers to independent variables and edges refers to the value of that independent variable. Each leaf corresponds to the value of a target variable (or label or class) on the basis of all independent variables lying on the path from root to leaf. Each of these internal nodes are computed recursively from root. First, from the root, the best attribute is chosen on the basis of information gain from that attribute. After selecting the first best attribute, the root will have children based on the attribute value. Then in each of the child nodes, this process will go on recursively. The stopping criteria can be either a threshold of information gain or the depth of the tree. The leaves of the tree contains the probability of each labels associated with the choice of independent attributes from root to the leave. The advantage of decision tree is that it is simple, understandable and interpretable. It can handle both numerical and categorical data and it can be used to multiple classification and regression problem. However, decision trees would suffer from overfitting issue. Sometimes, the constructed decision model is very complicated as well. It also become unstable sometimes due to the variation in the training datasets. Moreover, finding the optimal decision tree for a particular model is not straightforward process. In the scope of this work, we will use the standard decision tree library from the python scikit-learn package. It uses an optimized version of CART (classification and regression Trees) which is a variant of C4.5 decision tree.

3.3.2 Random Forest

Random Forest is an ensemble learning methods where multiple decision trees will be constructed from the training class and during testing, the class will be predicted that is the mode class predicted by all of the decision trees of that random forest. The intuition of random forest is to tackle the problem of overfitting suffered by the decision trees. In order to build the decision trees during the training time, example data of the training dataset will be randomly chosen along with replacement. During the computation of best splitting criteria, it would do that so from a randomly selected subset of the training example set. As a result, each tree tends to tackle the biasness and bagging along with random subset selection also ensures that individual trees have low correlation to each other leading to an ensemble that is both of low bias and low variance. In the scope of this work, we use the Random Forest library from scikit-learn package.

3.3.3 Fast and Frugal Tree

Fast and Frugal trees is a type of decision tree where a set of hierarchical rules is obtained from a training set. This tree has a depth of four usually and two branches in the each node. Each node represents a splitting attribute associated with a certain value or range. One of the branch denotes the subtree which will be used for further training if the condition is satisfied. The another branch is simply a leaf which denotes the cases if the condition is no satisfied. The leaf also contains the class information with associated probability. This type of tree is useful for quick interpretation of the decision models obtained from a training dataset. It is also useful for the dataset where number of key attributes is more or less, equal to the depth of the decision tree. In this work, we have implemented the Fast and Frugal tree from the scratch. We have used unsupervised discretization to select the best splitting attribute.

3.3.4 VFDT - Very Fast Decision Tree

Classic decision tree learners like ID3, C4.5 and CART assume that all training examples can be stored simultaneously in main memory, and are thus severely limited in the number of examples they can learn from. Disk-based decision tree learners like SLIQ and SPRINT assume the examples are stored on disk [10, 13], an learn by repeatedly reading them in sequentially (effectively once per level in the tree). While this greatly increases the size of usable training sets, it can become prohibitively expensive when learning complex trees (i.e., trees with many levels), and fails when datasets are too large to fit in the available disk space

Our goal is to design a decision tree learner for extremely large (potentially infinite) datasets. This learner should require each example to be read at most once, and only a small constant time to process it. This will make it possible to directly mine online data sources (i.e., without ever storing the examples), and to build potentially very complex trees with acceptable computational cost. We achieve this by noting with Catlett [2] and others that, in order to find the best attribute to test at a given node, it may be sufficient to consider only a small subset of the training examples that pass through that node. Thus, given a stream of examples, the first ones will be used to choose the root test; once the root attribute is chosen, the succeeding examples will be passed down to the corresponding leaves and used to choose the appropriate attributes there, and so on recursively.1 We solve the difficult problem of deciding exactly how many examples are necessary at each node by using a statistical result known as the Hoeffding bound (or additive Chernoff bound). Consider a real-valued random variable r whose range is R (e.g., for a probability the range is one, and for an information gain the range is $\log c$, where c is the number of classes). Suppose we have made n independent observations of this variable, and computed their mean $\bar{r}$. The Hoeffding

Table 1: The Hoeffding tree algorithm.

Inputs:	S	is a sequence of examples,
	$\mathbf{X}$	is a set of discrete attributes,
	$G(.)$	is a split evaluation function,
	δ	is one minus the desired probability of choosing the correct attribute at any given node.
Output:	HT	is a decision tree.

Procedure HoeffdingTree $(S, \mathbf{X}, G, \delta)$
Let HT be a tree with a single leaf l_1 (the root).
Let $\mathbf{X_1} = \mathbf{X} \cup \{X_\emptyset\}$.
Let $\overline{G}_1(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class in S.
For each class y_k
  For each value x_{ij} of each attribute $X_i \in \mathbf{X}$
    Let $n_{ijk}(l_1) = 0$.
For each example $(\mathbf{x}, y_k)$ in S
  Sort $(\mathbf{x}, y)$ into a leaf l using HT.
  For each x_{ij} in $\mathbf{x}$ such that $X_i \in \mathbf{X}_l$
    Increment $n_{ijk}(l)$.
  Label l with the majority class among the examples seen so far at l.
  If the examples seen so far at l are not all of the same class, then
    Compute $\overline{G}_l(X_i)$ for each attribute $X_i \in \mathbf{X}_l - \{X_\emptyset\}$ using the counts $n_{ijk}(l)$.
    Let X_a be the attribute with highest $\overline{G}_l$.
    Let X_b be the attribute with second-highest $\overline{G}_l$.
    Compute ϵ using Equation 1.
    If $\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ and $X_a \neq X_\emptyset$, then
      Replace l by an internal node that splits on X_a.
      For each branch of the split
        Add a new leaf l_m, and let $\mathbf{X_m} = \mathbf{X} - \{X_a\}$.
        Let $\overline{G}_m(X_\emptyset)$ be the $\overline{G}$ obtained by predicting the most frequent class at l_m.
        For each class y_k and each value x_{ij} of each attribute $X_i \in \mathbf{X_m} - \{X_\emptyset\}$
          Let $n_{ijk}(l_m) = 0$.
Return HT.

Figure 1. The VFDT Algorithm [5]

bound states that, with probability $1 - \delta$, the true mean of the variable is at least $\bar{r} - \epsilon$, where

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}} \tag{1}$$

The Hoeffding bound has the very attractive property that it is independent of the probability distribution generating the observations. The price of this generality is that the bound is more conservative than distribution-dependent ones (i.e., it will take more observations to reach the same δ and ϵ). Let $G(X_i)$ be the heuristic measure used to choose test attributes (e.g., the measure could be information gain as in C4.5, or the Gini index as in CART). Our goal is to ensure that, with high probability, the attribute chosen using

n examples (where n is as small as possible) is the same that would be chosen using infinite examples. Assume G is to be maximized, and let X_a be the attribute with highest observed $\bar{G}$ after seeing n examples, and X_b be the second-best attribute. Let $\Delta\bar{G} = \bar{G}(Xa) - \bar{G}(Xb) \geq 0$ be the difference between their observed heuristic values. Then, given a desired δ, the Hoeffding bound guarantees that X_a is the correct choice with probability $1 - \delta$ if n examples have been seen at this node and $\Delta\bar{G} > \epsilon^2$ In other words, if the observed $\Delta\bar{G} > \epsilon$ then the Hoeffding bound guarantees that the true $\Delta G \geq \Delta\bar{G} - \epsilon > 0$ with probability $1 - \delta$, and therefore that X_a is indeed the best attribute with probability $1 - \delta$. This is valid as long as the G value for a node can be viewed as an average of G values for the examples at that node, as is the case for the measures typically used. Thus a node needs to accumulate examples from the stream until ϵ becomes smaller than $\Delta\bar{G}$. (Notice that ϵ is a monotonically decreasing function of n.) At this point the node can be split using the current best attribute, and succeeding examples will be passed to the new leaves. This leads to the Hoeffding tree algorithm, shown in pseudo-code in 1.

A key property of the Hoeffding tree algorithm is that it is possible to guarantee under realistic assumptions that the trees it produces are asymptotically arbitrarily close to the ones produced by a batch learner (i.e., a learner that uses all the examples to choose a test at each node). In other words, the incremental nature of the Hoeffding tree algorithm does not significantly affect the quality of the trees it produces. In this work, we have implemented vfdt by python completely from the scratch.

3.4 Tuning Method

As CART, Random Forest and VFDT have several parameters that can pose impact in the classification performance, parameter tuning is necessary before deploying the learners for classifications tasks. We have used differential evolution method for finding the sub-optimal parameters for the classifiers. It finds the optimal solution for a problem by iteratively searching the canditate solution on the basis of a given fitness function. First it generates an initial population of candidate solutions and then generate subsequent set of candidate solutions by applying mutation and crossover methods. After that, all the possible solutions will be tested against a fitness function and best set of solutions will be kept. This process is done iteratively for several pass and final set of best solution will be considered. However, it must be kept in mind that this process does not guarantee that the theoretical optimal solution has been found. In this work, we have implemented this process from the scratch.

4 Experiment Design

In this work, we will apply four data miners named CART, FFT, RF and VFDT on four software defect prediction datasets. These datasets will be fed to the miners incrementally. We would observe the performance of all four miners based on these datasets. The experiment setup is described below:

4.1 Dataset

Although, there are so many standard defect prediction datasets, we choose use datasets that came from different sequential committed version of the same source code. It will allow us to experiment with large set of examples and incremented examples over a period of time. Here is the descriptions of the datasets:

Name	Examples	Attributes
abinit	89303	29
lammps	41130	40
libmesh	24673	40
mdanalysis	11694	37

Table 1. Dataset Description

After choosing the datasets, training and test example set needs to be generated. However, stratification and n-fold cross validation are not an option because those examples in the datasets are needed to be fed into the miners in incremental order (i.e. 5%, 10%, 15%… of the data). This is why we randomly choose 80% of the examples for each dataset and keep the rest of the data for testing set without breaking the sequence of the data. Thus, we have generated 10 different set of training and testing example pairs for each dataset.

4.2 Testbed Setup

The work is performed in a traditional laptop PC. Here is the information of the execution environment.

OS	Ubuntu 18.04
CPU	Intel Core i7 8550U
Memory	16GB
Disk	SSD 240GB
Language	Python 3.6
Data Miner Library	Scikit-Learn 0.20

Table 2. Testbed Information

4.3 Parameter Tuning

As CART, Random Forest and VFDT have several parameters that can pose impact in the classification performance, parameter tuning is necessary before deploying the learners for classifications tasks. Here is the last of parameters that are needed to be tuned for the classifiers.

talk about de parameters.

1. **Number of estimators:** This parameter is only applicable to random forest classifier. This is the number of trees that will be used for classification task in the random forest.
2. **Tree depth:** This parameter denotes the maximal depth of the decision tree.
3. **Minimum sample leaves:** This denotes the minimum number of samples required to split an internal node.
4. **Minimum number of examples to split:** This denotes the minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least of this value in each of the left and right branches.
5. **Ties:** When two or more attributes have very similar Gs, potentially many examples will be required to decide between them with high confidence. This is presumably wasteful, because in this case it makes little difference which attribute is chosen. Thus VFDT can optionally decide that there is effectively a tie and split on the current best attribute if $\delta G < \epsilon < \tau$, where τ is a user-specified threshold.
6. **Frequency of Computation of G:** The most significant part of the time cost per example is recomputing G. It is inefficient to recompute G for every new example, because it is unlikely that the decision to split will be made at that specific point. Thus VFDT allows the user to specify a minimum number of new examples nmin that must be accumulated at a leaf before G is recomputed. This effectively reduces the global time spent on G computations by a factor of n_min, and can make learning with VFDT nearly as fast as simply classifying the training examples. Notice, however, that it will have the effect of implementing a smaller δ than the one specified by the user, because examples will be accumulated beyond the strict minimum required to choose the correct attribute with confidence $1 - \delta$. (This increases the time required to build a node, but our experiments show that the net effect is still a large speedup.) Because δ shrinks exponentially fast with the number of examples, the difference could be large, and the δ input to VFDT should be correspondingly larger than the target.

Parameter	Range	Applicable Classifiers
Estimator	(10, 100)	RF
Depth	(2, 10)	CART, RF, VFDT
Splits	(0, 1), (5, 500)	CART, RF, VFDT
Leaves	(0, 1)	CART, RF
τ	(.001, .99)	VFDT
n_min	(5, 500)	VFDT

4.4 Evaluation Criteria

The evaluation criteria are discussed below:

1. **Memory:** It denotes the maximum amount of primary memory used by the python program containing classifiers. In this work, it will be measured in Megabytes.
2. **Time:** It denotes tha amount of time taken by the program to train from the dataset and construct the decision model. In this work, it will be measured in milliseconds.
3. **Precision:** This is the ratio of all detected bugs that are true positives and detected bugs that are true and false positives.

$$precision = \frac{true\ positive}{true\ positive + false\ positive} \tag{2}$$

4. **Recall:** This is the ratio of all detected bugs that are true positives and detected bugs that are false negatives and true positives. Recall score is very critical in the scope of this work because if recall score is low, that will mean that the classifiers would provide many false positive bugs.

$$recall = \frac{true\ positive}{true\ positive + false\ negative} \tag{3}$$

5. **False Alarm:** This is the ratio of all detected bugs that are false positives and detected bugs that are false negatives and false positives.

$$false\ alarm = \frac{false\ positive}{false\ positive + true\ negative} \tag{4}$$

It is noteworthy that we are not considering *accuracy* as it does not provide a good view of the prediction quality. In case of imbalanced data, it also outputs misleading information.

5 Result Analysis

In this section we will discuss the finidings in the light of research questions mentioned in section I.

5.1 RQ1: How these four classifiers performs in large defect prediction datasets that will increment over time

We will try to observe how CART, RF, FFT and VFDT performs in terms of precision and recall. Before deploying the learners, we have performed parameter tuning and applied the parameters obtained at 25% dataset size for the whole dataset.

From the figure 2 and 2 we can see that, with increase of the examples in abinit dataset, precision score for VFDT does not fluctuate but for the other three learners, it fluctuates, mostly FFT. However, in case of recall score, VFDT performs far better and stable across all of the sizes for the dataset. For the other three learners, the score gets lower and lower when the size increases.

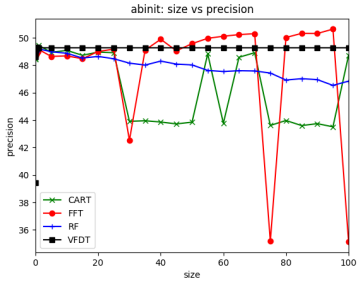

Figure 2. Size vs Precision for abinit dataset

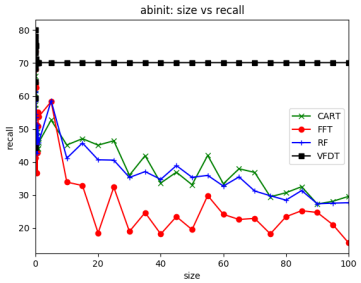

Figure 3. Size vs Recall for abinit dataset

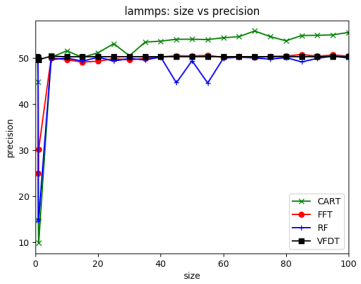

Figure 4. Size vs Precision for lammps dataset

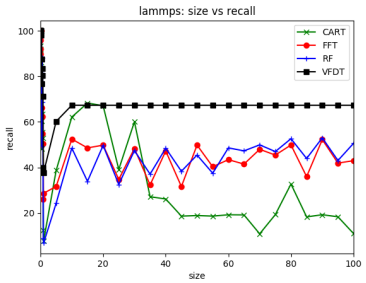

Figure 5. Size vs Recall for lammps dataset

From the figure 4 and 4 we can see that, with increase of the examples in lammps dataset, precision score for all the learners are similar and CART is performing slightly better than the others However, in case of recall score, VFDT slightly performs far better and stable across all of the sizes for the dataset. For the other three learners, the score fluctuates back and forth. However, CART performs the worst among the four learners.

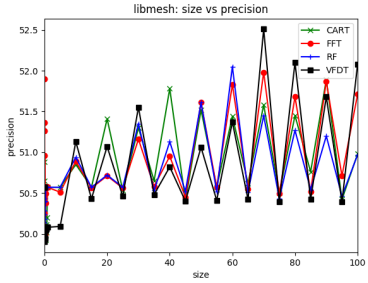

Figure 6. Size vs Precision for libmesh dataset

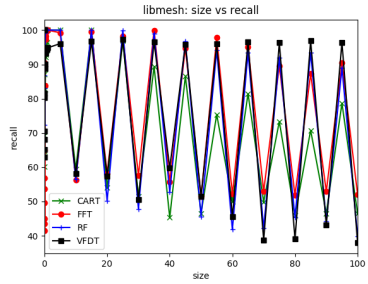

Figure 7. Size vs Recall for libmesh dataset

From the figure 6 and 6 we can see that, with increase of the examples in libmesh dataset, precision and recall score for all the learners are similar. However, it also shows us that with the increase of dataset size, the recall scores behaves very unstable, for all of the four learners.

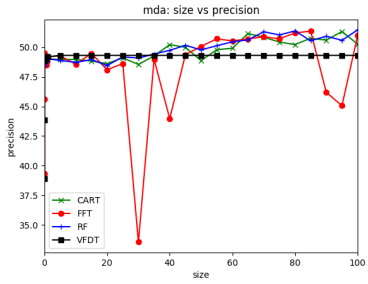

Figure 8. Size vs Precision for mdanalysis dataset

From the figure 8 and 8 we can see that, with increase of the examples in mdanalysis dataset, precision score for all the learners are similar and CART as well as FFT are performing slightly better than the others However, in case

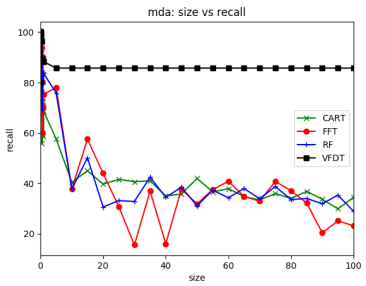

Figure 9. Size vs Recall for mdanalysis dataset

of recall score, VFDT performs far better and stable across all of the sizes for the dataset. For the other three learners, the score fluctuates back and forth a ittle bit. However, FFT performs the worst among the four learners.

From the figure 11, we can see that false alarm count for all the datasets are higher. It can also be seen that false alarm score is similar to recall score across all the datasets. Hence, we looked into the confusion matrix of the learners and observed that across all size, the number of True Negatives (TN) and False Negatives (FN) are higher than that of True Positives (TP) and False Positives (FP). Moreover, TN and FN are similar. Theoretically, if the precision score is around 50%, then recall score and false alarm score would become similar. This happens because, if precision is around 0.5, then TP will be similar to FP. So, that will mean, the value of $\frac{TP}{TP+FN}$ and $\frac{FP}{FP+FN}$ will be similar. Thus we can conclude that recall and false alarm will be similar.

In principle, the key observation from the analysis of four datasets are given below:

- VFDT performs better in recall score across all the dataset size. This can also be seen in the AUC score given in 10
- VFDT achieves slightly a lower precision score than the other three learners. This can also be seen in the AUC score given in 10
- Precision and recall scores of all the three learners except VFDT either goes up or down with the increase of data fed into them. For VFDT, it remains stable. As, VFDT reaches hoeffding bound and grows tree accordingly, with the increase of the data size, it's tree structure does not change. Hence, its precision and recall score also stay unchanged.
- As we performed tuning of the learners with 25% of the examples, it is obvious that optimal parameters for VFDT did not change while newer examples came. But with the arrival of newer examples, optimal set of parameters of CART and RF have changes. This is why their precision and recall score dropped.
- the precision score also tells us that all the learner performed marginally better than a random guess.

dataset	learner	x-axis	y-axis	AUC score
abinit	CART	size	precision	4606.96
abinit	FFT	size	precision	4816.09
abinit	RF	size	precision	4785.69
abinit	VFDT	size	precision	4926.43
abinit	CART	size	recall	3769.01
abinit	FFT	size	recall	2632.5
abinit	RF	size	recall	3620.71
abinit	VFDT	size	recall	7014.96
lammps	CART	size	precision	5246.65
lammps	FFT	size	precision	4969.28
lammps	RF	size	precision	4855.38
lammps	VFDT	size	precision	5026.41
lammps	CART	size	recall	3091.19
lammps	FFT	size	recall	4338.67
lammps	RF	size	recall	4336.18
lammps	VFDT	size	recall	6653.4
libmesh	CART	size	precision	5097.07
libmesh	FFT	size	precision	5096.04
libmesh	RF	size	precision	5087.96
libmesh	VFDT	size	precision	5090.76
libmesh	CART	size	recall	6864.43
libmesh	FFT	size	recall	7576.23
libmesh	RF	size	recall	7265.65
libmesh	VFDT	size	recall	7368.89
mda	CART	size	precision	4979.35
mda	FFT	size	precision	4835.72
mda	RF	size	precision	4996.91
mda	VFDT	size	precision	4929.88
mda	CART	size	recall	3928.77
mda	FFT	size	recall	3691.77
mda	RF	size	recall	3908.43
mda	VFDT	size	recall	8585.01

Figure 10. Size vs Recall for mdanalysis dataset

- High rate of false alarm is a genuine concern. It is also associated with the fact of precision score not getting better than the random guess.

5.2 RQ2: How hyper-parameters changes as the data size changes

From the analysis in previous section, it is obvious that when the learner faces substantial amount of newer examples, their prediction performance goes down. Consequently, it remains to be seen whether further parameter tuning improves their performance or not. Moreover, we understood that, if we want to decrease the false alarm, we need to increase the precision score. So we first tried parameter tuning to increase the f1 score with very high iteration count. As it is time

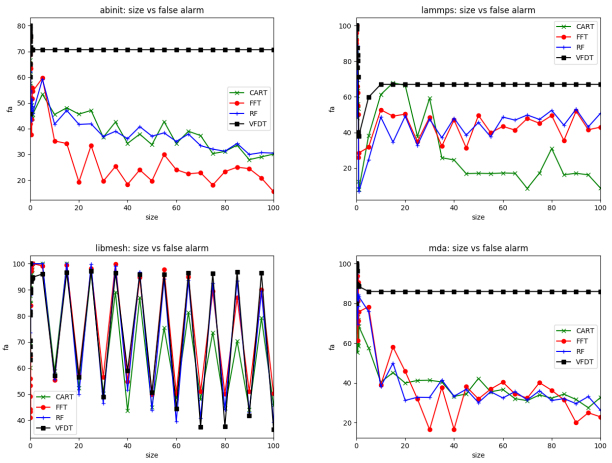

Figure 11. Data Size vs Time

consuming, we just applied parameter tuning for only CART on 25% of the datasets. We initially not to choose RF, FFT or VFDT because applying parameter tuning on them is time consuming due to the facts mentioned below:

1. We have already applied all the learners on all the datasets across 40 split size with 10 random sample of it. Still, we could not manage to get a decent f1 score. So, there might be a possibility that, we won't find any better precision and recall at the same time.
2. FFT is inherently time consuming for discretization on more than 25 attributes. It is also built from scratched and not as optimized as scikit-learn CART
3. RF needs to build at least 10 estimators and thus at least 10 times expensive than CART
4. VFDT can be much faster than CART. However, parameter tuning with n_min and τ can be time consuming as well.

After running the differential evolution algorithm, we found out that the best precision score we got is 0.56 from *lammps* dataset and worst we got is 0.48 from *abinit* dataset. After that, we ran the DE once again but with a goal to maximize the difference between recall and false alarm rate. We found out a maximum difference of 0.07 in *lammps* dataset.

Keeping this in mind, we perform another three parameter tuning while the datasets finished processing 50%, 75% and 100% of the total examples. We observed that precision score does not vary significantly across the dataset size from all the learners, hence in our differential evolution method, we tried to optimize the parameters on the basis of recall score. Here is the summary of the findings:

It is obvious from Figure 12, 13, 14, 15 that, for most of the cases recall score could have been improved if further tuning was deployed after seeing a substantial amount of newer examples. However, in case of learning for a large

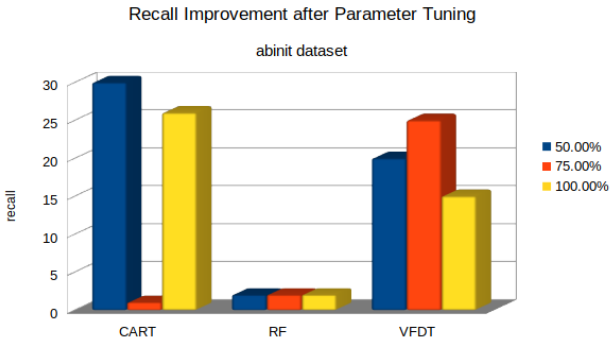

Figure 12. Size vs Recall for mdanalysis dataset

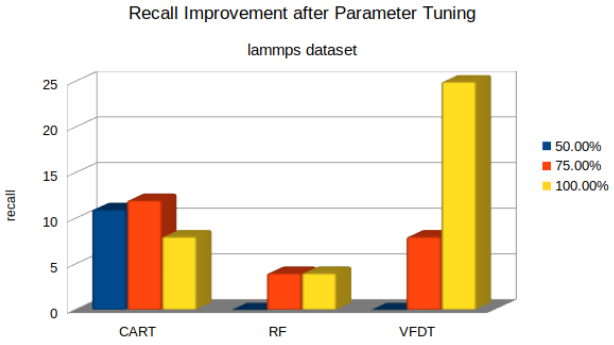

Figure 13. Size vs Recall for mdanalysis dataset

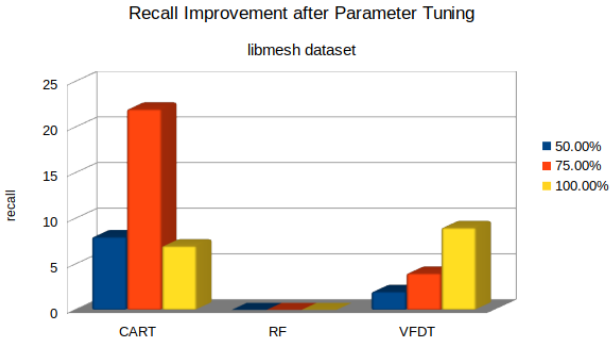

Figure 14. Size vs Recall for mdanalysis dataset

stream of data periodically, it is really inefficient to apply tuning each time the dataset learns.

5.3 RQ3: How much computation resources would be used by these four learners

It is also important to observe the computational resources being used by the learners because, with the increase of data size, the need for the learners to become cheap in terms of time and memory is paramount. Here, the findings from our analysis is presented below.

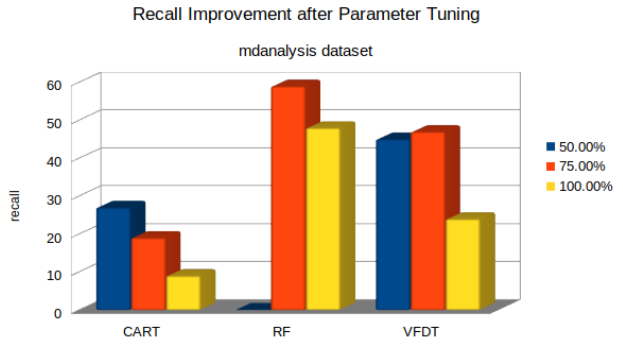

Figure 15. Size vs Recall for mdanalysis dataset

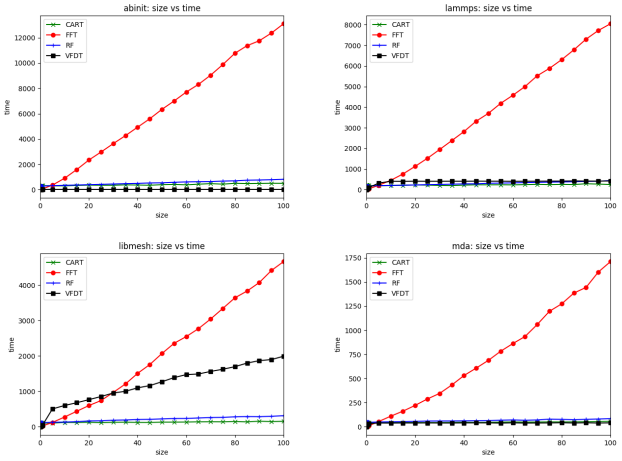

Figure 16. Data Size vs Time

From the above fig 16, we can see that FFT is very expensive in terms of computation time. Apart from libmesh dataset, it is also seen that CART, RF and VFDT all takes similar time to operate. However, such comparison between VFDT, CART and RF is not conclusive as the dataset is very small compared to the available primary memory in the testbed system. Hence, we needed to deploy the learner with a very big dataset that is similar to the size of primary memory of the computer. Unfortunately, such big dataset is not available for software defect prediction. Hence, we collected particle acceleration dataset from UCI Machine Learning dataset repository. The size of the dataset is around 8Gb of size. It contains 110000000 rows and 29 columns. All the class labels are either boolean and attributes are continuous. From our run, here is the outcome. From the table 3 above, it is obvious that VFDT scales in terms of both computation time and memory usage if the dataset size is substantially large.

5.4 Revisiting Baseline Criteria:

Our initital motivation is to observe whether stream based VFDT can do better than the offline classifiers. Hence, based on the observation above, here is our observation from the aspect of the aforementioned baseline.

Learner	Time	Memory
CART	332 sec	6.1 GB
RF	425 sec	6.1 GB
VFDT	34 sec	271 MB

Table 3. Computation Resource Usage for Very Large Dataset

1. **Useful:** In terms of precision, no learner is useful as those are not better than random guess significantly. However, in terms of recall, VFDT have outperformed others.
2. **Reasonable:** VFDT has performed no worse than the other established learners in this case.
3. **Stable:** VFDT has shown much more stable result than the other three
4. **Robust:** Across different datasets and splits, VFDT has shown more robust performance than the other three.
5. **Cheap:** VFDT is computationally cheaper than the others.
6. **Streaming:** VFDT can handle large datasets sequentially and it does not need to load the whole set of examples to primary memory.
7. **Tunable:** Parameter tuning yielded better recall rates for VFDT than the others across all the sizes.

6 Threats to Validity

1. **Sampling Bias:** Like every classification experiment, our work is also subject to have some bias in sampling training and test data. Although we have sampled 10 disjoint training and testing datasets for each dataset, it could have been better if the sampling number was higher
2. **Random Bias:** Some of the learners in our experiments utilized random value generation such as RF. Rerunning such classifiers multiple times could given us a less biased outputs.
3. **Parameter Bias:** In differential evolution algorithm, we have used some parameters to find the best parameters for the learners. So inherently, the tuning method also suffers from some degree of biasness induced by the parameters in the tuning algorithms.
4. **Code Optimization:** Our developing from scratch programs for VFDT and FFT are not as optimized as the CART and RF from scikit-learn package. So our experiment for computational resource could have been bias free if we had built the RF and CART from the scratch as well.

5. **Dataset:** The experiment we performed is biased on the four datasets. It should also be conducted on other large defect datasets to get a more neutral perception.
6. **Cross Validation:** The performance of the learners could have been better observed if cross-validation was performed. However, as the data is ordered, it was not possible.

7 Future Work Direction and Conclusion

Despite the fact that, VFDT has gained in all the baseline criteria we looked into so far, it has also opened several future work directions. Firstly, the experiment should also be performed in other large software defect datasets to gain even better understanding. Second, the issue of high rate of false alarm should be probed. Moreover, using multi-threading and multi-processing, VFDT can be made much more faster as well. Finally, C-VFDT [9] - which is a variant of VFDT can be much more useful in the defect prediction scenario as it can change its tree structure due to any potential distribution change in large dataset - can be deployed in defect prediction and the performance should be observed.

References

[1] Victor R Basili, Lionel C. Briand, and Walcélio L Melo. 1996. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on software engineering* 22, 10 (1996), 751–761.

[2] Nicolas Bettenburg, Meiyappan Nagappan, and Ahmed E Hassan. 2012. Think locally, act globally: Improving defect and effort prediction models. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*. IEEE Press, 60–69.

[3] Lionel C Briand, Jürgen Wüst, and Hakim Lounis. 2001. Replicated case studies for investigating quality factors in object-oriented designs. *Empirical software engineering* 6, 1 (2001), 11–58.

[4] Thomas G Dietterich. 2000. An experimental comparison of three methods for constructing ensembles of decision trees: Bagging, boosting, and randomization. *Machine learning* 40, 2 (2000), 139–157.

[5] Pedro Domingos and Geoff Hulten. 2000. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 71–80.

[6] Len Erlikh. 2000. Leveraging legacy system dollars for e-business. *IT professional* 2, 3 (2000), 17–23.

[7] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. 2012. A systematic literature review on fault prediction performance in software engineering. *IEEE Transactions on Software Engineering* 38, 6 (2012), 1276–1304.

[8] Ahmed E Hassan. 2009. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Computer Society, 78–88.

[9] Geoff Hulten, Laurie Spencer, and Pedro Domingos. 2001. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 97–106.

[10] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. 1996. SLIQ: A fast scalable classifier for data mining. In *International Conference on Extending Database Technology*. Springer, 18–32.

[11] Tim Menzies, Andrew Butcher, Andrian Marcus, Thomas Zimmermann, and David Cok. 2011. Local vs. global models for effort estimation and defect prediction. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 343–351.

[12] Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2014. Cross-project defect prediction models: L'union fait la force. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*. IEEE, 164–173.

[13] J Quinlan and R Cameron-Jones. 1995. Oversearching and layered search in empirical learning. *breast cancer* 286 (1995), 2–7.

[14] Emad Shihab. 2012. *An exploration of challenges limiting pragmatic software defect prediction.* Ph.D. Dissertation.

[15] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. 2007. Predicting defects for eclipse. In *Predictor Models in Software Engineering, 2007. PROMISE'07: ICSE Workshops 2007. International Workshop on*. IEEE, 9–9.