

**Minor Project Report
ON**

Parameter Tuning using Differential Evolution

SUBMITTED BY

Arjun Rajpal	(2K14/SE/21)
Dushyant Rathore	(2K14/SE/29)
Manav Middha	(2K14/SE/43)

Under the supervision of

Dr. RUCHIKA MALHOTRA

Assistant Professor

Dept. of Computer Science, Delhi Technological University



ACKNOWLEDGEMENT

“The successful completion of any task would be incomplete without acknowledging the people who made it all possible and whose constant guidance and encouragement secured us the success.”

First of all, we are grateful to the Almighty for establishing us to complete this minor project. We are grateful to Dr. Ruchika Malhotra, Assistant Professor (Department of Computer Science and Engineering), Delhi Technological University (Formerly Delhi College of Engineering), New Delhi and all other faculty members of our department, for their astute guidance, constant encouragement and sincere support for this project work.

We owe a debt of gratitude to our guide, Dr. Ruchika Malhotra, CSE Department for incorporating in us the idea of a creative Minor Project, helping us in undertaking this project and also for being there whenever we needed her assistance.

We also place on record, our sense of gratitude to one and all, who directly or indirectly have lent their helping hand in this venture. We feel proud and privileged in expressing my deep sense of gratitude to all those who have helped me in presenting this project.

Last but never the least, we thank our parents for always being with us, in every sense.

SUPERVISOR CERTIFICATE

This is to certify that **Arjun Rajpal (2K14/SE/021)**, **Dushyant Rathore (2K14/SE/29)**, **Manav Middha (2K14/SE/43)**, the bonafide students of Bachelor of Technology in Software Engineering of Delhi Technological University (formerly Delhi College of Engineering), New Delhi of 2014–2018 batch have completed their minor project entitled “Parameter Tuning using Differential Evolution” under the supervision of Dr. Ruchika Malhotra (Assistant Professor).

It is further certified that the work done in this dissertation is a result of candidates' own efforts.

I wish them all success in their life.

Date: _____

Dr. Ruchika Malhotra

Assistant Professor

Delhi Technological University

(Formerly Delhi College of Engineering)

Shahbad, Daulatpur, Bawana Road, Delhi – 110042

CONTENTS

1. Chapter 1: Introduction	4
2. Chapter 2: Literature Survey	7
3. Chapter 3: Proposed Work.....	113
4. Chapter 4: Experiments and Results.....	38
5. Chapter 5: Conclusion	47
6. Chapter 6: References.....	48

ABSTRACT

Parameter Tuning is a process in which one or more parameters of a device or model are adjusted upwards or downwards to achieve an improved or a specified result.

Software Defect Prediction involves detecting any flaw or imperfection in a software work product or a software process.

Data miners have been widely used in software engineering to generate defect predictors from static code measures. Static code defect predictors show better performance compared to manual methods. However, one of the essential components of data mining is setting the tunings that control the miner.

We seek simple, automatic, and effective method for finding tunings that give optimum value for the desired goal.

Our methodology consisted of experimentation with different data sets on which we ran Differential Evolution as an optimizer to explore the tuning space Then we tested the tunings using test data.

The results were astonishing. Contrary to our expectations, we found that tuning was remarkably simple i.e it only required few attempts to obtain excellent results.

We conclude that atleast for defect prediction, it is no longer sufficient to just run a data miner and present the result *without* conducting a tuning optimization study. The implication for other kinds of analytics is now an open and pressing issue.

Chapter 1: Introduction

1.1 Organization

In Chapter 2, we discuss the various pieces of work that we've studied and analysed in order to get a better grasp of the problem.

In Chapter 3, we discuss the explanation of the workflow, the various equations involved, pseudocode of the solution proposed, etc.

In Chapter 4, we plot graphs to help us visualize the dataset and solution model. We also analyse the algorithms used with the help of parameters such as accuracy, precision, etc.

In Chapter 5, we finally provide our conclusion by critically analysing the algorithms used in terms of their significance and the results seen. We explain why our work is a possible efficient solution to the problem of Parameter Tuning. We also discuss what can be the future improvements in this field.

1.2 Background

The ultimate goal of machine learning is to make a machine system that can automatically build models from data without requiring tedious and time consuming human involvement. One of the difficulties is that learning algorithms require the programmer to set parameters before you use the models (or at least to set constraints on those parameters). The goal, is usually to set those parameters to so optimal values that enable you to complete a learning task in the best way possible. Thus, tuning an algorithm or machine learning technique, can be simply thought of as process which one goes through in which they optimize the parameters that impact the model in order to enable the algorithm to perform the best.

For example, If you take a machine learning algorithm for clustering like KNN, you will note that you, as the programmer, must specify the number of K's in your model (or centroids), that are used. How do you do this? You tune the model. There are many ways that you can do this. One of these can be trying many different values of K for a model, and looking to understand how the inter and intra group error as you vary the number of K's in your model.

Data analysts are given a large amount of data to analyse. Many researchers are now turning towards automatic data miners. These learners automatically generate numbers that help in tuning.

1.3 Research Objective

In this report it is shown that using learners along with default parameters is not good enough. Therefore, tuning the learners to get the best parameters on the basis on performance measures like precision and f-measures is essential.

This report deals with Differential Evolution^[9] tuning method. Impacts of both the methods on defect predictors have been investigated. It is shown that differential evolution is as effective as other available methods while being faster than them.

The specific conclusion in this report is related to defect predictors but the techniques discussed can be extended to various fields like data science and software engineering. The goal is to comment how quickly tuning can be performed without incurring excessive CPU costs.

Chapter 2: Literature Survey

2.1 Defect Prediction

Defect prediction is a mechanism which can be effectively employed in predicting the defective classes or the fault prone classes in early development phases of software lifecycle. With the passage of time defect prediction^[10] has become a very important area of research. As the development of software progresses, the cost of maintenance increases manifold. The cost of correcting an error is 1000 times in the testing phase as compared to in the requirement phase. Since the available time and resources are limited, it is very crucial to identify the defective classes in the early development phases of software lifecycle^[6].

Any method that focusses on testing certain parts can miss out on errors in other parts so some sampling policy should be used to explore the system. One of the sampling policy is defect predictors learned from static attributes^[3]. Given defect attributes, data miners can find out where or when error is most likely to occur. This technique is faster and less labour intensive than manual code reviews.

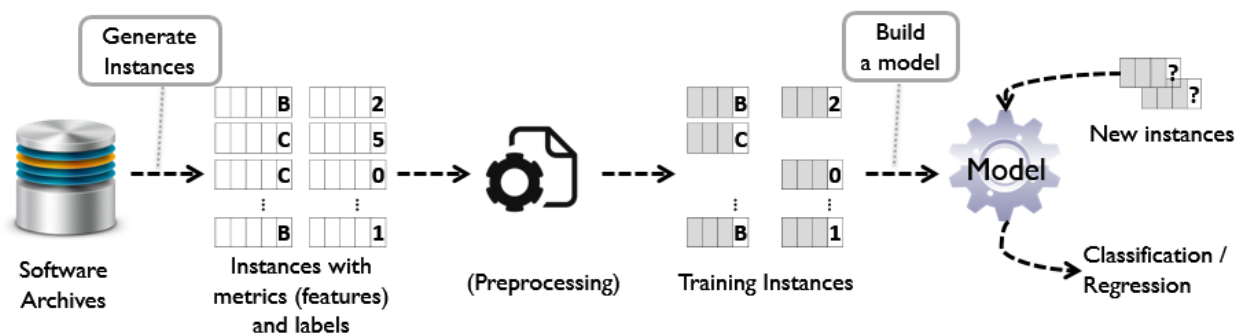


Figure 2.1: Defect Prediction using Machine Learning

2.2 Differential Evolution

Differential evolution (DE) is a method that optimizes a problem by iteratively trying to improve a candidate solution with regard to a given measure of quality. The Differential Evolution algorithm^[7] involves maintaining a population of candidate solutions subjected to iterations of recombination, evaluation, and selection. The recombination approach involves the creation of new candidate solution components based on the weighted difference between two randomly selected population members added to a third population member. This perturbs population members relative to the spread of the broader population. In conjunction with selection, the perturbation effect self-organizes the sampling of the problem space, bounding it to known areas of interest.

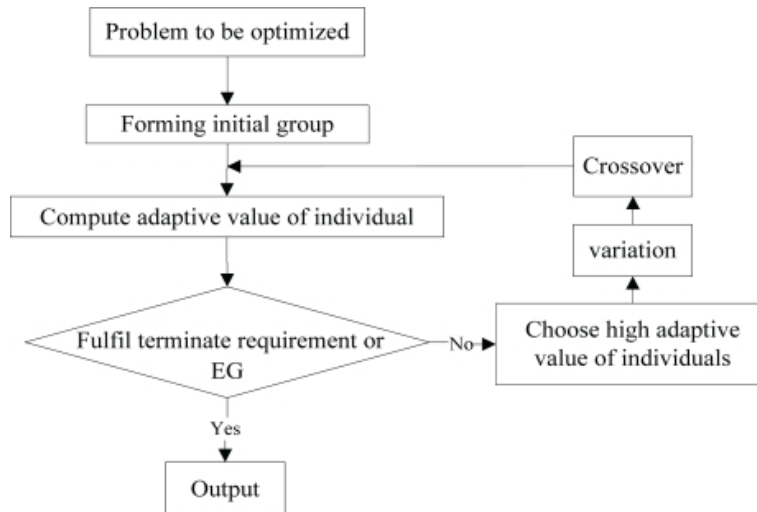


Figure 2.2: Differential Evolution flowchart

DE is an optimization technique which iteratively modifies a population of candidate solutions to make it converge to an optimum value of your function. First initialize candidate solutions randomly. Then at each iteration and for each candidate solution x , do the following:

1. Produce a trial vector: $v = a + (b - c) / 2$, where a, b, c are three distinct candidate solutions picked randomly among your population.
2. Randomly swap vector components between x and v to produce v' . At least one component from v must be swapped.
3. Replace x in your population with v' only if it is a better candidate (i.e. it better optimise your function).

2.3 Random Forest Algorithm

It is a classification and regression technique. It operates by constructing a multitude of decision trees at training time and outputting the mode or the mean of the classes.

Random Forest^[8] is a collection of decision trees. Decision trees can overfit but random forest prevents it i.e. that is prevents overfitting.

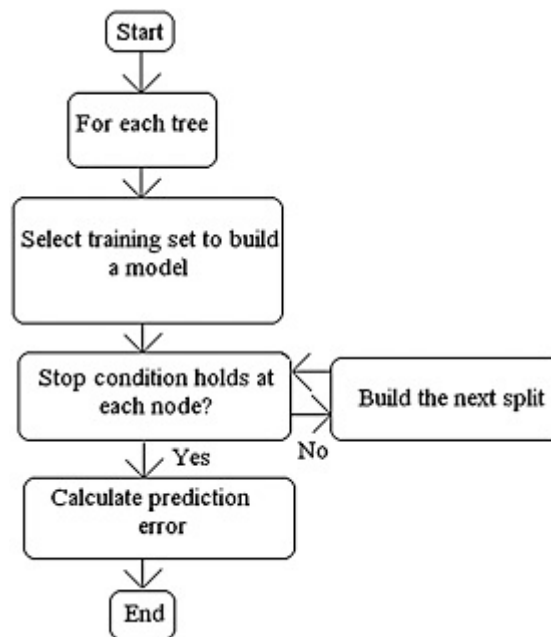


Figure 2.3: Random Forest flowchart

Advantages

1. High Accuracy.
2. Runs efficiently on large data bases.
3. Handles thousands of input variables without variable deletion.
4. Gives estimates of what variables are important in the classification.
5. Generates an internal unbiased estimate of the generalization error as the forest building progresses.
6. Provides effective methods for estimating missing data.
7. Maintains accuracy when a large proportion of the data are missing.
8. Provides methods for balancing error in class population unbalanced data sets.

Random Forest is very effective in eliminating noise in the model input data. Given a long list of input variables and a potentially sparse dataset, it is very likely that any predictive model will discover spurious relationships between those inputs and the chosen target variable. This results in overfitting and the model does not generalize

well enough to future input it has not seen.

Because Random Forest builds many trees using a subset of the available input variables and their values, it inherently contains some underlying decision trees that omit the noise generating variable/feature(s). In the end, when it is time to generate a prediction, a vote among all the underlying trees takes place and the majority prediction value wins.

2.4 Classification And Regression Trees Algorithm

Decision Trees are commonly used in data mining with the objective of creating a model that predicts the value of a target (or dependent variable) based on the values of several input (or independent variables).

Classification Trees: Where the target variable is categorical and the tree is used to identify the "class" within which a target variable would likely fall into.

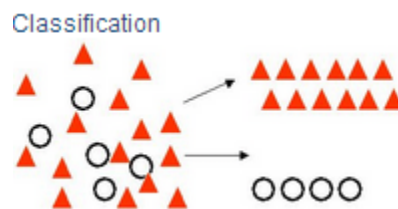


Figure 2.4: Classification Trees

Regression Trees: Where the target variable is continuous and tree is used to predict it's value.



Figure 2.5: Regression Trees

The main elements of CART (and any decision tree algorithm) are:

1. Rules for splitting data at a node based on the value of one variable.
2. Stopping rules for deciding when a branch is terminal and can be split no more.
3. Finally, a prediction for the target variable in each terminal node.

2.5 Logistics Regression Algorithm

Logistic regression predicts the probability of an outcome that can only have two values (i.e. a dichotomy). The prediction is based on the use of one or several predictors (numerical and categorical). A linear regression is not appropriate for predicting the value of a binary variable for two reasons.

- A linear regression will predict values outside the acceptable range (e.g. predicting probabilities outside the range 0 to 1).
- Since the dichotomous experiments can only have one of two possible values for each experiment, the residuals will not be normally distributed about the predicted line.

On the other hand, a logistic regression produces a logistic curve, which is limited to values between 0 and 1. Logistic regression is similar to a linear regression, but the curve is constructed using the natural logarithm of the “odds” of the target variable, rather than the probability. Moreover, the predictors do not have to be normally distributed or have equal variance in each group.

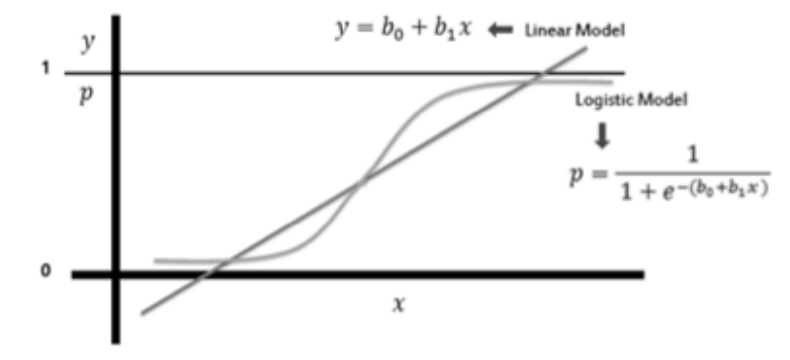


Figure 2.6: Comparison of linear regression model and logistic regression model

In the logistic regression the constant (b_0) moves the curve left and right and the slope (b_1) defines the steepness of the curve. By simple transformation, the logistic regression equation can be written in terms of an odds ratio.

Finally, taking the natural log of both sides, we can write the equation in terms of log-odds (logit) which is a linear function of the predictors. The coefficient (b_1) is the amount the logit (log-odds) changes with a one unit change in x.

$$\frac{p}{1-p} = \exp(b_0 + b_1x)$$

As mentioned before, logistic regression can handle any number of numerical and/or categorical variables.

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1x$$

There are several analogies between linear regression and logistic regression. Just as ordinary least square regression is the method used to estimate coefficients for the best fit line in linear regression, logistic regression uses maximum likelihood estimation (MLE) to obtain the model coefficients that relate predictors to the target. After this initial function is estimated, the process is repeated until LL (Log Likelihood) does not change significantly.

$$p = \frac{1}{1 + e^{-(b_0 + b_1x_1 + x_2b_2 + \dots + b_px_p)}}$$

Chapter 3: Proposed Work

Defect prediction^[5] is a mechanism which can be effectively employed in predicting the defective classes or the fault prone classes in early development phases of software lifecycle. With the passage of time defect prediction has become very a very important area of research.

In order to understand the relative merits of differential evolution for defect prediction, we performed the following study.

3.1 Tuning Parameters:

Our DE and grid search explored the parameter space of Figure 3.1. Specifically, since Tantithamthavorn et al. divide each tuning range into 5 bins (if applicable), we also use the same policy here. For example, we pick values [50,75,100,125,150] for n estimators. Other parameters grid will generate in the same way. As to why we used the “Tuning Range” shown in Figure 3.1, and not some other ranges, we note that (1) those ranges included the defaults; (2) the results shown below show that by exploring those ranges, we achieved large gains in the performance of our defect predictors. This is not to say that larger tuning ranges might not result in greater improvements.

Learner Name	Parameters	Default	Tuning Range	Description
CART	threshold	0.5	[0,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	max_depth	None	[1, 50]	The maximum depth of the tree.
Random Forests	threshold	0.5	[0.01,1]	The value to determine defective or not.
	max_feature	None	[0.01,1]	The number of features to consider when looking for the best split.
	max_leaf_nodes	None	[1,50]	Grow trees with max_leaf_nodes in best-first fashion.
	min_sample_split	2	[2,20]	The minimum number of samples required to split an internal node.
	min_samples_leaf	1	[1,20]	The minimum number of samples required to be at a leaf node.
	n_estimators	100	[50,150]	The number of trees in the forest.

Figure 3.1: List of parameters tuned by this paper

3.2 Data:

Our defect data, shown in Figure 3.2 comes from PROMISE^[1] repository. This data pertains to open source Java systems: ant, camel, ivy, jedit, log4j, lucene, poi, synapse, velocity and xerces. We selected these data sets since they have at least three consecutive releases (where release i+1 was built after release i). This will allow us to build defect predictors^[2] based on the past data and then predict (test) defects on future version projects, which will be a more practical scenario.

More specifically, when tuning a learner:

- Release i was used for training a learner with tunings generated by grid search or differential evolution.
- During the search, each candidate has to be evaluated by some model, which we build using CART or Random Forests from release $i+1$.
- After grid search or differential evolution terminated, we tested the tunings found by those methods on CART or Random Forests applied to release $i+2$.
- For comparison purposes, CART and Random Forests were also trained (with default tunings) on releases i and $i+1$ then tested on release $i+2$.

Dataset	antV0	antV1	antV2	camelV0	camelV1	ivy	jeditV0	jeditV1	jeditV2
training (release i)	20 / 125	40 / 178	32 / 293	13 / 339	216 / 608	63 / 111	90 / 272	75 / 306	79 / 312
tuning (release $i+1$)	40 / 178	32 / 293	92 / 351	216 / 608	145 / 872	16 / 241	75 / 306	79 / 312	48 / 367
testing (release $i+2$)	32 / 293	92 / 351	166 / 745	145 / 872	188 / 965	40 / 352	79 / 312	48 / 367	11 / 492
Dataset	log4j	lucene	poiV0	poiV1	synapse	velocity	xercesV0	xercesV1	
training (release i)	34 / 135	91 / 195	141 / 237	37 / 314	16 / 157	147 / 196	77 / 162	71 / 440	
tuning (release $i+1$)	37 / 109	144 / 247	37 / 314	248 / 385	60 / 222	142 / 214	71 / 440	69 / 453	
testing (release $i+2$)	189 / 205	203 / 340	248 / 385	281 / 442	86 / 256	78 / 229	69 / 453	437 / 588	

Figure 3.2 Data used in this case study. Fractions denote defects / total. Eg: the top left dataset has 20 defective classes out of 125 total.

3.3 Algorithm Description:

Differential Evolution

The working of DE algorithm is very simple. Consider you need to optimize (minimize, for example) $\sum X_i^2$ (sphere model) within a given range, say $[-100, 100]$. We know that the minimum value is 0. Let's see how DE works.

DE is a population-based algorithm. And for each individual in the population, a fixed number of chromosomes will be there (imagine it as a set of human beings and chromosomes or genes in each of them). Let us explain DE w.r.t above function

We need to fix the population size and the number of chromosomes or genes (named as parameters). For instance, let's consider a population of size 4 and each of the individual has 3 chromosomes (or genes or parameters). Let's call the individuals R1, R2, R3, R4.

Step 1: Initialize the population

We need to randomly initialise the population within the range $[-100, 100]$

G1 G2 G3 objective function value R1 -> $|-90| \ 2 \ | \ 1| \Rightarrow 8105$ R2 -> $| \ 7 \ | \ 9 \ | \ -50| \Rightarrow 2630$ R3 -> $| \ 4 \ | \ 2 \ | \ -9.2| \Rightarrow 104.64$ R4 -> $| \ 8.5 \ | \ 7 \ | \ 9 \ | \Rightarrow 202.25$

objective function value is calculated using the given objective function. In this case, it's $\sum X_i^2$. So for R1, objective function value will be $-90^2 + 2^2 + 1^2 = 8105$. Similarly, it is

found for all.

Step 2: Mutation

Fix a target vector, say for example R1 and then randomly select three other vectors (individuals) say for example, R2, R3, R4 and performs mutation. Mutation is done as follows,

$$\text{Mutant Vector} = R2 + F(R3-R4)$$

(vectors can be chosen randomly, need not be in any order). **F (scaling factor/mutation constant) within range [0,1]** is one among the few control parameters DE is having. In simple words, it describes how different the mutated vector becomes. Let's keep $F = 0.5$.

$$| 7 | 9 | -50 | + 0.5 * | 4 | 2 | -9.2 | + | 8.5 | 7 | 9 |$$

Now performing Mutation will give the following Mutant Vector:

$$\text{MV} = | 13.25 | 13.5 | -50.1 | \Rightarrow 2867.82$$

Step 3: Crossover

Now that we have a target vector(R1) and a mutant vector MV formed from R2, R3 & R4, we need to do a crossover. Consider R1 and MV as two parents and we need a child from these two parents. Crossover is done to determine how much information is to be taken from both the parents. It is controlled by **Crossover rate(CR)**. Every gene/chromosome of the child is determined as follows, a random number between 0 & 1 is generated, if it is greater than CR, then inherit a gene from target(R1) else from mutant(MV).

Let's set $CR = 0.9$. Since we have 3 chromosomes for individuals, we need to generate 3 random numbers between 0 and 1. Say for example, those numbers are 0.21, 0.97, 0.8 respectively. First and last are lesser than CR value, so those positions in the child's vector will be filled by values from MV and second position will be filled by gene taken from target(R1).

$$\text{Target} \rightarrow | -90 | 2 | 1 | \quad \text{Mutant} \rightarrow | 13.25 | 13.5 | -50.1 |$$

$$\begin{aligned} \text{random num} - 0.21, & \Rightarrow \text{Child} \rightarrow | 13.25 | -- | -- | \quad \text{random num} - 0.97, \Rightarrow \text{Child} \rightarrow | 13.25 | 2 | -- | \\ \text{random num} - 0.80, & \Rightarrow \text{Child} \rightarrow | 13.25 | 2 | -50.1 | \quad \text{Trial vector/child vector} \rightarrow | 13.25 | 2 | -50.1 | \\ & \Rightarrow 2689.57 \end{aligned}$$

Step 4: Selection

Now we have child and target. Compare the objective function of both, see which is smaller (minimization problem). Select that individual out of the two for next generation

$$R1 \rightarrow | -90 | 2 | 1 | \Rightarrow 8105 \quad \text{Trial vector/child vector} \rightarrow | 13.25 | 2 | -50.1 | \Rightarrow 2689.57$$

Clearly, the child is better so replace target(R1) with the child. So the new population will become

$$\begin{aligned} G1 \quad G2 \quad G3 \quad \text{objective fn value} \quad R1 \rightarrow & | 13.25 | 2 | -50.1 | \Rightarrow 2689.57 \quad R2 \rightarrow | 7 | 9 | -50 | \Rightarrow 2500 \\ R3 \rightarrow & | 4 | 2 | -9.2 | \Rightarrow 104.64 \quad R4 \rightarrow | -8.5 | 7 | 9 | \Rightarrow 202.25 \end{aligned}$$

This procedure will be continued either till the number of generations desired has reached or till we get our desired value.

Random Forest

The implementation of Random Forest as given in sklearn is as follows :

```
class sklearn.ensemble.RandomForestClassifier (n_estimators=10, criterion='gini',
max_depth=None, min_samples_split=2, min_samples_leaf=1,
min_weight_fraction_leaf=0.0, max_features='auto', max_leaf_nodes=None,
min_impurity_split=1e-07, bootstrap=True, oob_score=False, n_jobs=1,
random_state=None, verbose=0, warm_start=False, class_weight=None)
```

Cart

```
def cart(a, b, c, d, candidate):
```

```
ct = DecisionTreeClassifier(max_depth=candidate["tunings"][4],
min_samples_split=candidate["tunings"][2], min_samples_leaf=candidate["tunings"][3],
max_features=candidate["tunings"][1], min_impurity_split=candidate["tunings"][0])
```

```
ct.fit(a, b)
```

```
pred = ct.predict(c)
```

3.4 Codes:

1) Untuned Logistics Regression, Random Forest & Cart

```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import precision_score, f1_score
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from prettytable import PrettyTable
```

```
dataset = ['antV0', 'antV1', 'antV2', 'camelV0', 'camelV1', 'ivy', 'jeditV0', 'jeditV1', 'jeditV2',
'log4j', 'lucene', 'poiV0', 'poiV1', 'synapse', 'velocity', 'xercesV0', 'xercesV1']
```

```
precision_cart = []
precision_rf = []
precision_lr = []
```

```
fscore_cart = []
fscore_rf = []
fscore_lr = []
```

CART

```
def cart(a, b, c, d):
    ct = DecisionTreeClassifier(min_impurity_split=0.5, min_samples_split=2,
min_samples_leaf=1, max_features=None, max_depth=None)
    ct.fit(a, b)
    pred = ct.predict(c)
    # print "Predicted Matrix : " + str(pred)

    p = precision_score(d, pred, average='weighted')
    # p = accuracy_score(d, pred)

    # print p
    precision_cart.append(p * 100)

    f = f1_score(d, pred, average='weighted')
    fscore_cart.append(f * 100)
```

Random Forest

```
def random_forest(a, b, c, d):
    rf = RandomForestClassifier(min_impurity_split=0.5, min_samples_split=2,
min_samples_leaf=1, n_estimators=100, max_features=None, max_leaf_nodes=None)
    rf.fit(a, b)
    pred = rf.predict(c)
    # print "Predicted Matrix : " + str(pred)

    p = precision_score(d, pred, average='weighted')
    # p = accuracy_score(d, pred)

    # print p
    precision_rf.append(p * 100)

    f = f1_score(d, pred, average='weighted')
    fscore_rf.append(f * 100)
```

Logistic Regression

```
def logistic_regression(a, b, c, d):
    lr = LogisticRegression()
    lr.fit(a, b)
    pred = lr.predict(c)
    # print "Predicted Matrix : " + str(pred)
```

```

p = precision_score(d, pred, average='weighted')
# p = accuracy_score(d, pred)

# print p
# print "\n"
precision_lr.append(p * 100)

f = f1_score(d, pred, average='weighted')
fscore_lr.append(f * 100)

def perform_calculation(loc_train, loc_test):

    f1 = pd.read_csv(loc_train, delimiter=",")
    df1 = pd.DataFrame(f1)

    f2 = pd.read_csv(loc_test, delimiter=",")
    df2 = pd.DataFrame(f2)

    # Training Set
    X_Train_DF = df1.ix[:, 3:23]
    X_Train = X_Train_DF.values.astype(float)
    y_Train = np.asarray(list(df1["bug"]))
    # print X_Train
    # print Y_Train

    # Testing Set
    X_Test_DF = df2.ix[:, 3:23]
    X_Test = X_Test_DF.values.astype(float)
    y_Test = np.asarray(list(df2["bug"]))
    # print X_Test
    # print Y_Test

    cart(X_Train, y_Train, X_Test, y_Test)
    random_forest(X_Train, y_Train, X_Test, y_Test)
    logistic_regression(X_Train, y_Train, X_Test, y_Test)

def calculate_all():

    # antV0
    perform_calculation("../combined_dataset_modified/antV0/1.csv",
                        "../combined_dataset_modified/antV0/3.csv")

    # antV1
    perform_calculation("../combined_dataset_modified/antV1/2.csv",
                        "../combined_dataset_modified/antV1/4.csv")

```

```

# antV2
perform_calculation("../combined_dataset_modified/antV2/3.csv",
"../combined_dataset_modified/antV2/5.csv")

# camelV0
perform_calculation("../combined_dataset_modified/camelV0/1.csv",
"../combined_dataset_modified/camelV0/3.csv")

# camelV1
perform_calculation("../combined_dataset_modified/camelV1/2.csv",
"../combined_dataset_modified/camelV1/4.csv")

# ivy
perform_calculation("../combined_dataset_modified/ivy/1.csv",
"../combined_dataset_modified/ivy/3.csv")

# jeditV0
perform_calculation("../combined_dataset_modified/jeditV0/1.csv",
"../combined_dataset_modified/jeditV0/3.csv")

# jeditV1
perform_calculation("../combined_dataset_modified/jeditV1/2.csv",
"../combined_dataset_modified/jeditV1/4.csv")

# jeditV2
perform_calculation("../combined_dataset_modified/jeditV2/3.csv",
"../combined_dataset_modified/jeditV2/5.csv")

# log4j
perform_calculation("../combined_dataset_modified/log4j/1.csv",
"../combined_dataset_modified/log4j/3.csv")

# lucene
perform_calculation("../combined_dataset_modified/lucene/1.csv",
"../combined_dataset_modified/lucene/3.csv")

# poiV0
perform_calculation("../combined_dataset_modified/poiV0/1.csv",
"../combined_dataset_modified/poiV0/3.csv")

# poiV1
perform_calculation("../combined_dataset_modified/poiV1/2.csv",
"../combined_dataset_modified/poiV1/4.csv")

# synapse
perform_calculation("../combined_dataset_modified/synapse/1.csv",
"../combined_dataset_modified/synapse/3.csv")

```

```

# velocity
perform_calculation("../combined_dataset_modified/velocity/1.csv",
"../combined_dataset_modified/velocity/3.csv")

# xercesV0
perform_calculation("../combined_dataset_modified/xercesV0/1.csv",
"../combined_dataset_modified/xercesV0/3.csv")

# xercesV1
perform_calculation("../combined_dataset_modified/xercesV1/2.csv",
"../combined_dataset_modified/xercesV1/4.csv")

# Print Precision table for Cart, Random Forest and Logistic Regression
sequence_precision = ["Dataset", "Cart", "RF "]

t_precision = PrettyTable(sequence_precision)

for i in range(0, len(dataset)):
    t_precision.add_row([dataset[i],precision_cart[i], precision_rf[i]])

print "Table for precision of Untuned Learner"
print str(t_precision) + "\n"

# Print fscore table for Cart, Random Forest
sequence_fscore = ["Dataset", "Cart", "RF"]

t_fscore = PrettyTable(sequence_fscore)

for i in range(0, len(dataset)):
    t_fscore.add_row(
        [dataset[i], fscore_cart[i], fscore_rf[i]])

print "Table for fscore of Untuned Learner"
print t_fscore

return precision_cart,precision_rf, precision_lr,fscore_cart,fscore_rf,fscore_lr

if __name__ == "__main__":
    calculate_all()

```

II) Tuned Random Forest using Precision & F measure as Goal state one by one

```

from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import precision_score

```

```

import pandas as pd
import numpy as nump
import math
from data import data
import random
from prettytable import PrettyTable

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

# np, f, cr, life, Goal, noOfParameters = 10, 0.75, 0.3, 5, [], 0 #input
# sBest = [] #output

global global_best_score
global_best_score = []
global global_best
global_best= []

# Runs Random Forest on the dataset using parameters present in candidate
def random_forest(a, b, c, d, candidate):
    # print "Candidate in Random Forest : " + str(candidate)
    # print "Threshold : " + str(candidate["tunings"][0])
    # print "Max feature : " + str(candidate["tunings"][1])
    # print "Max leaf nodes : " + str(candidate["tunings"][2])
    # print "Min sample split : " + str(candidate["tunings"][3])
    # print "Min samples leaf : " + str(candidate["tunings"][4])
    # print "Max no of estimators : " + str(candidate["tunings"][5])

    rf = RandomForestClassifier(min_impurity_split=candidate["tunings"][0],
max_features=candidate['tunings'][1],
                                max_leaf_nodes=candidate['tunings'][2],
min_samples_split=candidate['tunings'][3],
                                min_samples_leaf=candidate['tunings'][4],
n_estimators=candidate['tunings'][5])
    rf.fit(a, b)
    pred = rf.predict(c)
    # print "Predicted Matrix : " + str(pred)
    p = precision_score(d, pred, average='weighted')

```

```

return p

# Retrieves the appropriate dataset from readDataset function present in data.py and passes
# the Training and Testing dataset to Random Forest function
# Retrieves the appropriate dataset from readDataset function present in data.py and passes
# the Training and Testing dataset to Random Forest function
def score(candidate, datasets):

    df1, df2 = data.readDataset(datasets)

    # Training Set
    X_Train_DF = df1.ix[:, 3:23]
    X_Train = X_Train_DF.values.astype(float)
    Y_Train = numpy.asarray(list(df1["bug"]))
    # print X_Train
    # print Y_Train

    # Testing Set
    X_Test_DF = df2.ix[:, 3:23]
    X_Test = X_Test_DF.values.astype(float)
    Y_Test = numpy.asarray(list(df2["bug"]))
    # print X_Test
    # print Y_Test

    p = random_forest(X_Train, Y_Train, X_Test, Y_Test, candidate)

    return p

def score_test(candidate, datasets):

    df1, df2 = data.readDataset_for_testing(datasets)

    # Training Set
    X_Train_DF = df1.ix[:, 3:23]
    X_Train = X_Train_DF.values.astype(float)
    Y_Train = numpy.asarray(list(df1["bug"]))
    # print X_Train
    # print Y_Train

    # Testing Set
    X_Test_DF = df2.ix[:, 3:23]
    X_Test = X_Test_DF.values.astype(float)
    Y_Test = numpy.asarray(list(df2["bug"]))
    # print X_Test
    # print Y_Test

```

```

p = random_forest(X_Train, Y_Train, X_Test, Y_Test, candidate)

return p

# Initialises the population such that each parameter of each candidate
# in the population gets a random value from the parameter's valid range
def initialisePopulation(np,noOfParameters):
    population = []

    for i in range(0,np):
        candidate = {}
        tunings = []
        # tunings.append(numpy.random.uniform(algoParameters[0]['low'],
        algoParameters[0]['high']))

        threshold = numpy.random.uniform(algoParameters[0]['low'], algoParameters[0]['high'])
        # print "Threshold for index " + str(i) + " : " + str(threshold)
        tunings.append(threshold)

        max_feature = int(numpy.random.uniform(algoParameters[1]['low'],
        algoParameters[1]['high']))
        # print "Max Feature selected for index " + str(i) + " : " + str(max_feature)
        tunings.append(max_feature)

        max_leaf_nodes = int(numpy.random.uniform(algoParameters[2]['low'],
        algoParameters[2]['high']))
        # print "Max Leaf Nodes for index " + str(i) + " : " + str(max_leaf_nodes)
        tunings.append(max_leaf_nodes)

        min_sample_split = int(numpy.random.uniform(algoParameters[3]['low'],
        algoParameters[3]['high']))
        # print "Min sample split for index " + str(i) + " : " + str(min_sample_split)
        tunings.append(min_sample_split)

        min_samples_leaf = int(numpy.random.uniform(algoParameters[4]['low'],
        algoParameters[4]['high']))
        # print "Min samples leaf for index " + str(i) + " : " + str(min_samples_leaf)
        tunings.append(min_samples_leaf)

        n_estimators = int(numpy.random.uniform(algoParameters[5]['low'],
        algoParameters[5]['high']))
        # print "n_estimators for index " + str(i) + " : " + str(n_estimators) + "\n"
        tunings.append(n_estimators)

        # population[i]['tunings']
        # print tunings[4]
        candidate['tunings'] = tunings

```



```

    candidate['score'] = 0

    population.append(candidate)

    # print "Population"
    # print population

    return population

# Returns three random candidates from a population of size 10 such that
# neither of them is equal to the target candidate i.e old
def threeOthers(pop,old,index):

    # while(1):
    three = range(0,10,1)
    three.remove(index)
    three = random.sample(three, 3) # Array Formation

    # print pop[three[0]]['tunings']
    # print pop[three[0]]['tunings']
    # print pop[three[0]]['tunings']

    return pop[three[0]]['tunings'], pop[three[1]]['tunings'], pop[three[2]]['tunings']

# Returns True if the newly computed population is different than the old generation
otherwise False
def improve(population, oldGeneration):
    if population != oldGeneration:
        return True
    else:
        return False

# Compares the precision score of each candidate in the population and
# returns the candidate with the highest precision score as the Best Solution
def getBestSolution(population):
    max = -1
    bestSolution = {}

    for i in range(0,len(population)):
        scores = population[i]['score']
        # print scores

        if scores > max:
            max = scores
            bestSolution = population[i]

```

```
return bestSolution
```

```
# Creates a mutant whose each parameter value is either taken from the target candidate(old)
or is
```

```
# computed using  $(a+f*(b-c))$  depending on value of cr
```

```
def extrapolate(old, pop, cr, f, noOfParameters, index, dataset):
```

```
    a, b, c = threeOthers(pop, old, index) # index is for the target row
```

```
    newf = []
```

```
    # print "\nThe other 3 selected rows for index " + str(index) + " : "
```

```
    # print a
```

```
    # print b
```

```
    # print c
```

```
    for i in range(0, noOfParameters):
```

```
        x = numpy.random.uniform(0, 1)
```

```
        # print "Random number for comparison with cr : " + str(x)
```

```
        if cr < x:
```

```
            # print "Old tuning Value for index " + str(index) + " : " + (str(old['tunings'][i]))
```

```
            newf.append(old['tunings'][i])
```

```
        elif type(old['tunings'][i]) == bool:
```

```
            newf.append(not old['tunings'][i])
```

```
        else:
```

```
            lo = algoParameters[i]["low"]
```

```
            hi = algoParameters[i]["high"]
```

```
            value = a[i] + (f * (b[i] - c[i]))
```

```
            # print "Value before trim : " + str(value)
```

```
            if i != 0:
```

```
                mutant_value = int(max(lo, min(value, hi)))
```

```
            else:
```

```
                mutant_value = max(lo, min(value, hi))
```

```
            # print "Mutant Value : " + str(mutant_value)
```

```
            newf.append(mutant_value)
```

```
dict_mutant = {'tunings': newf}
```

```
score_mutant = score(dict_mutant, dataset)
```

```
score_original = score(old, dataset)
```

```

# print "Original Score : " + str(score_original)
# print "Mutant Score : " + str(score_mutant)

global global_best
global global_best_score

if score_mutant > score_original:
    global_best_score.append(score_mutant * 100)
    global_best.append({'score': score_mutant * 100, 'tunings': newf})

else:
    global_best_score.append(score_original)
    global_best.append({'score': score_original * 100, 'tunings': old["tunings"]})

# print global_best

newCandidate = {'score': 0, 'tunings': newf}
# print newCandidate
return newCandidate

# Performs Differential Evolution
def DE(np, f, cr, life, noOfParameters, dataset):

    population = initialisePopulation(np, noOfParameters) # Intial population formation

    while life > 0:

        global global_best
        global_best = []
        global global_best_score
        global_best_score = []

        for i in range(0, np):
            extrapolate(population[i], population, cr, f, noOfParameters, i, dataset)

        # print "Global Best : "
        # print global_best

        oldPopulation = []
        globalPopulation = []

        for row in population:
            oldPopulation.append(row['tunings'])

        for row in global_best:
            globalPopulation.append(row['tunings'])

```

```

# print "Old Population :"
# print oldPopulation
#
# print "Global Population :"
# print globalPopulation

if oldPopulation != globalPopulation:
    population = global_best
    # print population
else:
    life -= 1

s_Best = getBestSolution(global_best)

return s_Best

# Sets the valid range for each parameter of the machine learning algorithm
algoParameters = [{'low': 0.01, 'high': 1}, {'low': 1, 'high': 20}, {'low': 2, 'high': 50}, {'low': 2, 'high': 20},
                  {'low': 1, 'high': 20}, {'low': 50, 'high': 150}]

# Invokes DE

all_data_precision_rf = []

print "Precision Random Forest"

def calculate():
    for i in range(0, 17):
        dataset = i
        parameters = DE(10, 0.75, 0.3, 5, 6, dataset)
        score_p = (score_test(parameters, dataset) * 100)
        print color.BOLD + color.CYAN + "\nBest Parameters for Random Forest in dataset ",
        str(dataset + 1), " are ", str(parameters) + color.END
        print color.BOLD + color.GREEN + "Precision Score : " + str(score_p) + color.END +
        "\n"
        all_data_precision_rf.append(score_p)

if __name__ == "__main__":
    calculate()

```

III) Tuned CART using Precision & F measure as Goal state one by one

```

from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

```

```

from sklearn.metrics import precision_score
import pandas as pd
import numpy as nump
import math
from data import data
import random
from prettytable import PrettyTable

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

# np, f, cr, life, Goal, noOfParameters = 10, 0.75, 0.3, 5, [], 0 #input
# sBest = [] #ouput

global global_best_score
global_best_score = []
global global_best
global_best= []

# Runs Random Forest on the dataset using parameters present in candidate
def cart(a, b, c, d, candidate):
    # print "\nCandidate in Cart : " + str(candidate)
    # # value_threshold = nump.random.uniform(0.01, 1)
    # # print "Threshold :" + str(value_threshold)
    # print "Threshold :" + str(candidate["tunings"][0])
    # print "Max feature : " + str(candidate["tunings"][1])
    # print "Min sample split : " + str(candidate["tunings"][2])
    # print "Min samples leaf : " + str(candidate["tunings"][3])
    # print "Max depth : " + str(candidate["tunings"][4])

    ct = DecisionTreeClassifier(max_depth=candidate["tunings"][4],
min_samples_split=candidate["tunings"][2], min_samples_leaf=candidate["tunings"][3],
max_features=candidate["tunings"][1], min_impurity_split=candidate["tunings"][0])
    ct.fit(a, b)
    pred = ct.predict(c)
    # print "Predicted Matrix : " + str(pred)
    p = precision_score(d, pred, average='weighted')

```

```

return p

# Retrieves the appropriate dataset from readDataset function present in data.py and passes
# the Training and Testing dataset to Random Forest function
def score(candidate, dataset):

    df1, df2 = data.readDataset(dataset)

    # Training Set
    X_Train_DF = df1.ix[:, 3:23]
    X_Train = X_Train_DF.values.astype(float)
    Y_Train = numpy.asarray(list(df1["bug"]))
    # print X_Train
    # print Y_Train

    # Testing Set
    X_Test_DF = df2.ix[:, 3:23]
    X_Test = X_Test_DF.values.astype(float)
    Y_Test = numpy.asarray(list(df2["bug"]))
    # print X_Test
    # print Y_Test

    p = cart(X_Train, Y_Train, X_Test, Y_Test, candidate)

    return p

def score_test(candidate, dataset):

    df1, df2 = data.readDataset_for_testing(dataset)

    # Training Set
    X_Train_DF = df1.ix[:, 3:23]
    X_Train = X_Train_DF.values.astype(float)
    Y_Train = numpy.asarray(list(df1["bug"]))
    # print X_Train
    # print Y_Train

    # Testing Set
    X_Test_DF = df2.ix[:, 3:23]
    X_Test = X_Test_DF.values.astype(float)
    Y_Test = numpy.asarray(list(df2["bug"]))
    # print X_Test
    # print Y_Test

    p = cart(X_Train, Y_Train, X_Test, Y_Test, candidate)

```

```

return p

# Initialises the population such that each parameter of each candidate
# in the population gets a random value from the parameter's valid range
def initialisePopulation(np,noOfParameters):
    population = []

    for i in range(0,np):
        candidate = {}
        tunings = []
        # tunings.append(nump.random.uniform(algoParameters[0]['low'],
        algoParameters[0]['high']))

        threshold= nump.random.uniform(algoParameters[0]['low'], algoParameters[0]['high'])
        # print "Threshold for index " + str(i) + " : " + str(threshold)
        tunings.append(threshold)

        max_feature = int(nump.random.uniform(algoParameters[1]['low'],
        algoParameters[1]['high']))
        # print "Max Feature selected for index " + str(i) + " : " + str(max_feature)
        tunings.append(max_feature)

        min_sample_split = int(nump.random.uniform(algoParameters[2]['low'],
        algoParameters[2]['high']))
        # print "Max Leaf Nodes for index " + str(i) + " : " + str(min_sample_split)
        tunings.append(min_sample_split)

        min_samples_leaf = int(nump.random.uniform(algoParameters[3]['low'],
        algoParameters[3]['high']))
        # print "Min sample split for index " + str(i) + " : " + str(min_samples_leaf)
        tunings.append(min_samples_leaf)

        max_depth = int(nump.random.uniform(algoParameters[4]['low'],
        algoParameters[4]['high']))
        # print "Min samples leaf for index " + str(i) + " : " + str(max_depth)
        tunings.append(max_depth)

        # population[i]['tunings']
        # print tunings[4]
        candidate['tunings'] = tunings
        candidate['score'] = 0

    population.append(candidate)

# print "Population"
# print population

```

```

return population

# Returns three random candidates from a population of size 10 such that
# neither of them is equal to the target candidate i.e old
def threeOthers(pop,old,index):

    # while(1):
    three = range(0,10,1)
    three.remove(index)
    three = random.sample(three, 3) # Array Formation

    # print pop[three[0]]['tunings']
    # print pop[three[0]]['tunings']
    # print pop[three[0]]['tunings']

    return pop[three[0]]['tunings'], pop[three[1]]['tunings'], pop[three[2]]['tunings']

# Returns True if the newly computed population is different than the old generation
otherwise False
def improve(population, oldGeneration):
    if population != oldGeneration:
        return True
    else:
        return False

# Compares the precision score of each candidate in the population and
# returns the candidate with the highest precision score as the Best Solution
def getBestSolution(population):
    max = -1
    bestSolution = {}

    for i in range(0,len(population)):
        scores = population[i]['score']
        # print scores

        if scores > max:
            max = scores
            bestSolution = population[i]

    return bestSolution

# Creates a mutant whose each parameter value is either taken from the target candidate(old)
or is
# computed using  $(a+f*(b-c))$  depending on value of cr

```



```

def extrapolate(old, pop, cr, f, noOfParameters, index, dataset):
    a, b, c = threeOthers(pop, old, index) # index is for the target row

    newf = []

    # print "\nThe other 3 selected rows for index " + str(index) + " : "
    # print a
    # print b
    # print c

    for i in range(0, noOfParameters):

        x = numpy.random.uniform(0, 1)
        # print "Random number for comparison with cr : " + str(x)

        if cr < x:
            # print "Old tuning Value for index " + str(index) + " : " + (str(old['tunings'][i]))
            newf.append(old['tunings'][i])

        elif type(old['tunings'][i]) == bool:
            newf.append(not old['tunings'][i])
        else:
            lo = algoParameters[i]["low"]
            hi = algoParameters[i]["high"]
            value = a[i] + (f * (b[i] - c[i]))

            # print "Value before trim : " + str(value)

            if i != 0:
                mutant_value = int(max(lo, min(value, hi)))
            else:
                mutant_value = max(lo, min(value, hi))

            # print "Mutant Value : " + str(mutant_value)

            newf.append(mutant_value)

    dict_mutant = {'tunings': newf}
    score_mutant = score(dict_mutant, dataset)

    score_original = score(old, dataset)

    # print "Original Score : " + str(score_original)
    # print "Mutant Score : " + str(score_mutant)

    global global_best
    global global_best_score

```

```

if score_mutant > score_original:
    global_best_score.append(score_mutant * 100)
    global_best.append({'score': score_mutant * 100, 'tunings': newf})

else:
    global_best_score.append(score_original)
    global_best.append({'score': score_original * 100, 'tunings': old["tunings"]})

# print global_best

# newCandidate = {'score': 0, 'tunings': newf}
# print newCandidate
# return newCandidate

# Performs Differential Evolution
def DE(np, f, cr, life, noOfParameters, dataset):

    population = initialisePopulation(np, noOfParameters) # Intial population formation

    while life > 0:

        global global_best
        global_best = []
        global global_best_score
        global_best_score = []

        for i in range(0, np):
            extrapolate(population[i], population, cr, f, noOfParameters, i, dataset)

        # print "Global Best :"
        # print global_best

        oldPopulation = []
        globalPopulation = []

        for row in population:
            oldPopulation.append(row['tunings'])

        for row in global_best:
            globalPopulation.append(row['tunings'])

        # print "Old Population :"
        # print oldPopulation
        #
        # print "Global Population :"
        # print globalPopulation

```

```

    if oldPopulation != globalPopulation:
        population = global_best
        # print population
    else:
        life -= 1

    s_Best = getBestSolution(global_best)

return s_Best

# Sets the valid range for each parameter of the machine learning algorithm
algoParameters = [{'low': 0, 'high': 1}, {'low': 1, 'high': 20}, {'low': 2, 'high': 20}, {'low': 1,
'high': 20},
                  {'low': 1, 'high': 50}]

# Invokes DE

print "Precision Cart"

def calculate():
    all_data_precision_cart = []

    for i in range(0, 17):
        dataset = i
        parameters = DE(10, 0.75, 0.3, 5, 5, dataset)
        score_p = (score_test(parameters, dataset) * 100)
        # print color.BOLD + color.CYAN + "\nBest Parameters for Cart in dataset ",
str(dataset + 1), " are ", str(parameters) + color.END
        # print color.BOLD + color.GREEN + "Precision Score : " + str(score_p) + color.END
+ "\n"
        all_data_precision_cart.append(score_p)

    return all_data_precision_cart

if __name__ == "__main__":
    calculate()

```

IV) Plotting Graphs

```

import matplotlib.pyplot as plt
import lr_vs_rf_vs_cart
import DE_cart
import DE_cart_fscore
import DE_random_forest
import DE_random_forest_fscore

```

```

class color:
    PURPLE = '\033[95m'
    CYAN = '\033[96m'
    DARKCYAN = '\033[36m'
    BLUE = '\033[94m'
    GREEN = '\033[92m'
    YELLOW = '\033[93m'
    RED = '\033[91m'
    BOLD = '\033[1m'
    UNDERLINE = '\033[4m'
    END = '\033[0m'

# Untuned Models

precision_cart_untuned, precision_rf_untuned, precision_lr_untuned, fscore_cart_untuned,
fscore_rf_untuned, \
fscore_lr_untuned = lr_vs_rf_vs_cart.calculate_all()

print color.BOLD + color.RED + "Untuned Model" + color.END
print color.BOLD + color.CYAN + "Precision list for Untuned Cart" + color.END +
str(precision_cart_untuned)
print color.BOLD + color.CYAN + "Fscore list for Untuned Cart " + color.END +
str(fscore_cart_untuned)
print color.BOLD + color.CYAN + "Precision list for Untuned Random Forest" +
color.END + str(precision_rf_untuned)
print color.BOLD + color.CYAN + "Fscore list for Untuned Random Forest " + color.END
+ str(fscore_rf_untuned)

# Tuned Models

precision_cart_tuned = DE_cart.calculate()
fscore_cart_tuned = DE_cart.fscore.calculate()

precision_rf_tuned = DE_random_forest.calculate()
fscore_rf_tuned = DE_random_forest.fscore.calculate()

print color.BOLD + color.RED + "Tuned Model" + color.END
print color.BOLD + color.CYAN + "Precision list for Tuned Cart" + color.END +
str(precision_cart_tuned)
print color.BOLD + color.CYAN + "Fscore list for Tuned Cart " + color.END +
str(fscore_cart_tuned)
print color.BOLD + color.CYAN + "Precision list for Tuned Random Forest" + color.END
+ str(precision_rf_tuned)
print color.BOLD + color.CYAN + "Fscore list for Tuned Random Forest " + color.END +
str(fscore_rf_tuned)
print "\n"

```

```

# Graph Plotting

data = []
for i in range(0,17):
    data.append(i+1)

delta_cart_precision = []
for i in range(0,17):
    delta_cart_precision.append(abs(precision_cart_tuned[i] - precision_cart_untuned[i]))

plt.figure(1)
plt.title("Precision for Tuned vs Untuned Cart")
plt.plot(data, precision_cart_untuned, color='r', label='Untuned CART')
plt.plot(data, precision_cart_tuned, color='g', label='Tuned CART')
plt.legend()

plt.figure(2)
plt.title("Fscore for Tuned vs Untuned Cart")
plt.plot(data, fscore_cart_untuned, color='r', label='Untuned CART')
plt.plot(data, fscore_cart_tuned, color='g', label='Tuned CART')
plt.legend()

plt.figure(3)
plt.title("Delta in precision for Cart")
plt.plot(data, delta_cart_precision, color='r')
plt.legend()

plt.figure(4)
plt.title("Delta in fscore for Cart")
plt.plot(data, delta_cart_fscore, color='r')
plt.legend()

plt.figure(5)
plt.title("Precision for Tuned vs Untuned Random Forest")
plt.plot(data, precision_rf_untuned, color='r', label='Untuned Random Forest')
plt.plot(data, precision_rf_tuned, color='g', label='Tuned Random Forest')
plt.legend()

plt.figure(6)
plt.title("Fscore for Tuned vs Untuned Random Forest")
plt.plot(data, fscore_rf_untuned, color='r', label='Untuned Random Forest')
plt.plot(data, fscore_rf_tuned, color='g', label='Tuned Random Forest')
plt.legend()

plt.figure(7)
plt.title("Delta in precision for Random Forest")
plt.plot(data, delta_rf_precision, color='r')
plt.legend()

```

```
plt.figure(8)
plt.title("Delta in fscore for Random Forest")
plt.plot(data, delta_rf_fscore, color='r')
plt.legend()
```

```
plt.figure(9)
plt.title("Precision for Untuned Cart, Random Forest and Logistic Regression")
plt.plot(data, precision_cart_untuned, color='r', label='Untuned CART')
plt.plot(data, precision_rf_untuned, color='g', label='Untuned Random Forest')
plt.plot(data, precision_lr_untuned, color='b', label='Untuned Logistic Regression')
plt.legend()
```

```
plt.figure(10)
plt.title("Fscore for Untuned Cart, Random Forest and Logistic Regression")
plt.plot(data, fscore_cart_untuned, color='r', label='Untuned CART')
plt.plot(data, fscore_rf_untuned, color='g', label='Untuned Random Forest')
plt.plot(data, fscore_lr_untuned, color='b', label='Untuned Logistic Regression')
plt.legend()
```

```
plt.show()
```

3.4 Optimization Goals:

Our optimizers explore tuning improvements for precision and the F-measure, defined as follows. Let {A, B, C, D} denote the true negatives, false negatives, false positives, and true positives (respectively) found by a binary detector. Certain standard measures can be computed from A, B, C, D, as shown below. Note that for f-measure, the better scores are smaller while for all other scores, the better scores are larger.

$$pd = recall = D/(B+D)$$

$$pf = C/(A+C)$$

$$prec = precision = D/(D+C)$$

$$F = 2 * pd * prec / (pd + prec)$$

Chapter 4 – Experiments and Results

We propose to evaluate the performance of Machine Learning algorithms like Logistic Regression, CART (Classification And Regression Trees) and Random Forest on the basis of whether they were tuned or untuned before they were applied on the different datasets.

The objective was to determine the effect of tuning the parameters of the algorithms on Defect Prediction in terms of Precision and F-measure.

I) UNTUNED DATASETS

Each of the three algorithms - Logistics Regression, Random Forest and CART were applied on all the 17 datasets one by one by taking default values of their parameters. The Precision and F-measure was noted for each dataset.

A) PRECISION

Datasets	Logistics Regression	Random Forest	CART
antV0	81.99404	82.12890	82.12890
antV1	34.82052	25.0	25.0
antV2	45.54957	38.30409	43.75047
camelV0	40.01302	35.20200	35.81006
camelV1	44.19069	40.94668	39.41437
ivy	72.0	77.35	69.97899
jeditV0	31.35724	28.28278	37.60467
jeditV1	39.58333	36.82598	47.81746
jeditV2	90.90909	79.54545	92.72727
log4j	29.61348	7.99419	22.00978
lucene	23.83467	22.11611	24.33392
poiV0	9.21960	9.55334	34.85078
poiV1	47.11064	48.57635	56.35903
synapse	36.52034	36.56030	36.56030
velocity	29.27350	34.65811	36.55555
xercesV0	30.60640	29.34782	28.44961
xercesV1	29.64043	14.79973	16.17081

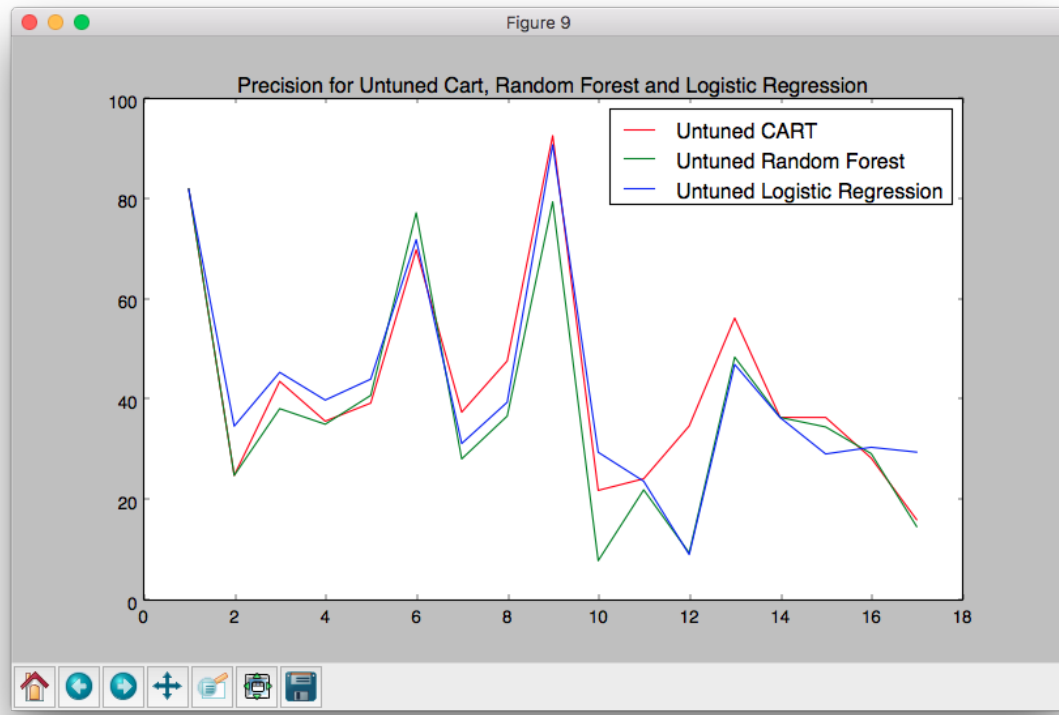


Figure 4.1: Precision for Untuned CART, Random Forest & Logistics Regression

B) F-MEASURE

Datasets	Logistics Regression	Random Forest	CART
antV0	68.875	86.16803	86.16803
antV1	36.85300	33.33333	33.33333
antV2	47.90314	43.51349	44.47823
camelV0	40.17245	39.66459	38.20472
camelV1	46.27693	44.88794	42.94473
ivy	60.41666	64.641241	55.07692
jeditV0	33.75733	35.788262	36.04078
jeditV1	40.90909	38.104838	49.96580
jeditV2	41.95804	70.707070	44.98834
log4j	10.68724	3.36276	10.02873
lucene	24.96998	24.98323	25.49327
poiV0	13.79928	14.56590	21.50814
poiV1	37.14868	57.46920	61.73726
synapse	38.87745	45.56791	45.56791
velocity	32.32905	36.88751	38.05422
xercesV0	31.19771	34.82035	31.87025
xercesV1	31.64147	18.30648	19.61072

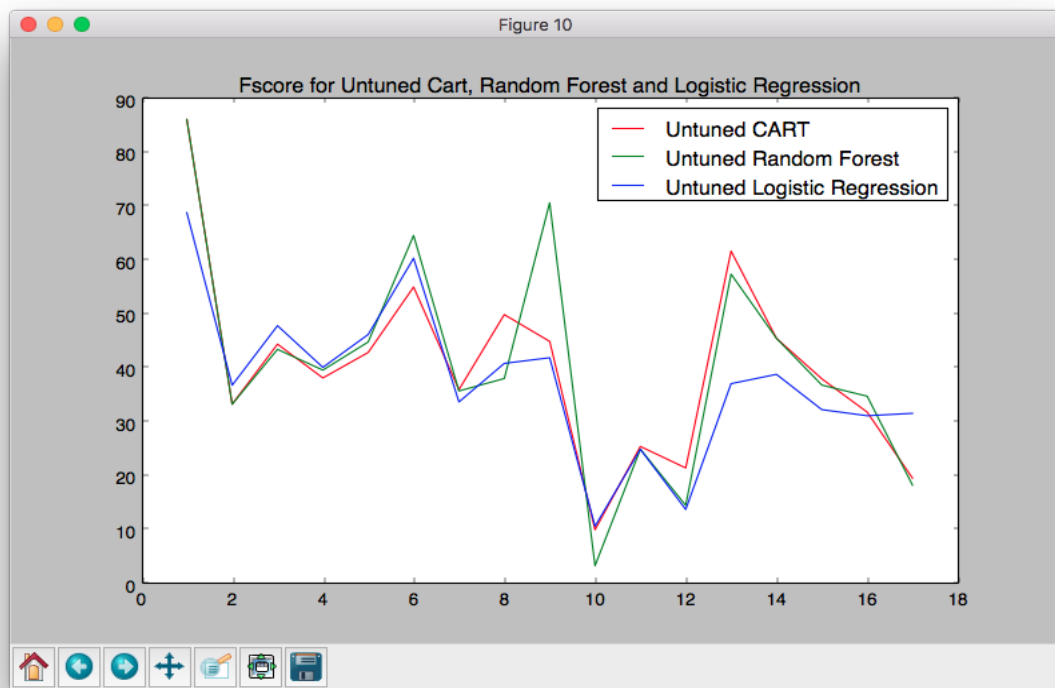


Figure 4.2: F-measure for Untuned CART, Random Forest & Logistics Regression

II) TUNED DATASETS

Differential Evolution is applied on Random Forest and CART one by one for each of the 17 datasets with the aim of optimising one of the performance parameters like Precision and F-measure. DE is executed twice for each dataset in case of each algorithms by taking Precision as optimising goal in one case and F-measure in the other.

A) RANDOM FOREST USING PRECISION AS OPTIMISING GOAL

Datasets	Precision	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ sample s_ leaf	n_estimators
antV0	82.12890	0.46754	6	42	10	13	94
antV1	25	0.46754	6	30	5	14	130
antV2	41.38699	0.76280	11	49	5	19	124
camelV0	23.97621	0.79167	9	44	7	11	119
camelV1	58.32742	0.01	27	45	2	5	90
ivy	74.54545	0.23476	13	21	6	4	148
jeditV0	36.21157	0.01	17	50	5	1	149
jeditV1	39.24557	0.21	7	51	5	1	127
jeditV2	92.99988	0.07	2	49	14	6	140
log4j	18.56332	0.39227	14	26	8	4	105
lucene	24.56772	0.19334	14	29	10	7	105

poiV0	37.77988	0.01	8	49	14	4	140
poiV1	51.16576	0.03063	10	29	8	4	115
synapse	36.56030	0.33090	9	3	11	9	133
velocity	32.56332	0.34522	14	36	8	2	105
xercesV0	37.50374	0.46991	16	41	19	9	123
xercesV1	15.48091	0.05803	15	50	2	14	50

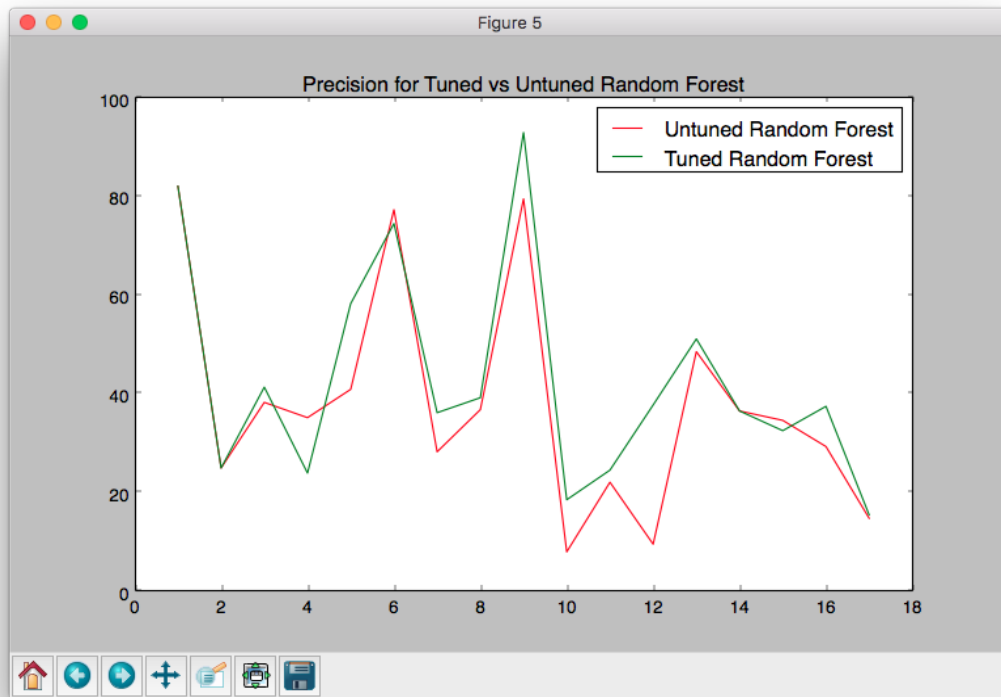


Figure 4.3: Precision for Tuned vs Untuned Random Forest

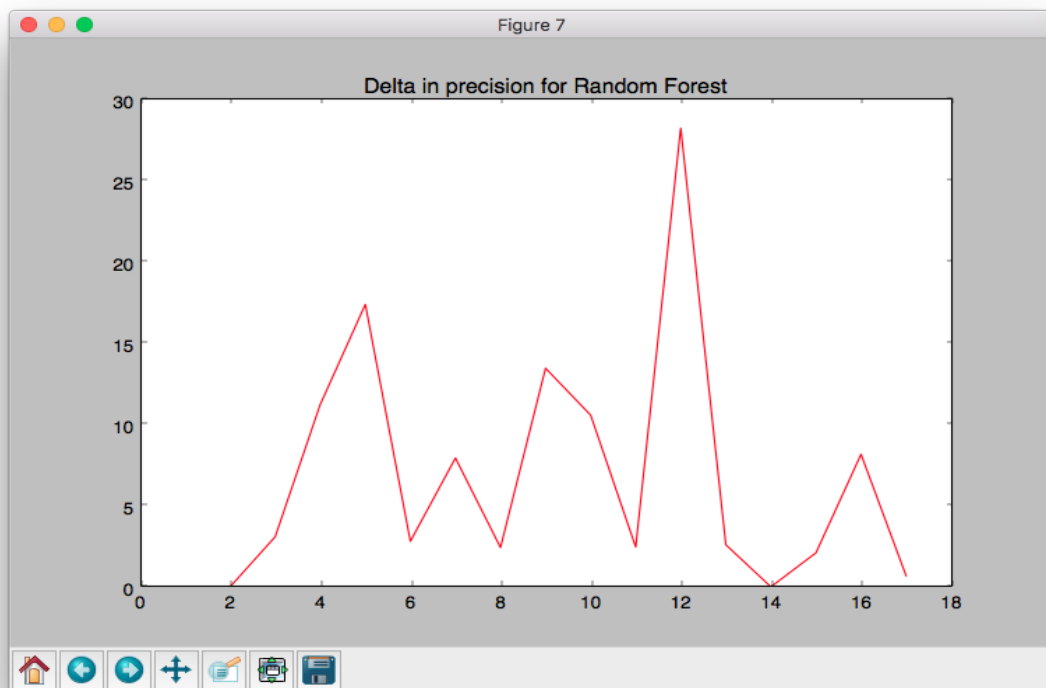


Figure 4.4: Delta in Precision for Random Forest

B) RANDOM FOREST USING F-MEASURE AS OPTIMISING GOAL

Dataset	F-Measure	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	86.16803	0.82704	6	7	2	2	143
antV1	33.33333	0.83924	8	26	8	19	106
antV2	40.23352	0.20340	1	12	7	12	94
camelV0	52.19029	0.89316	13	23	12	3	138
camelV1	48.42361	0.02943	1	12	12	4	53
ivy	77.64705	1	6	19	16	18	135
jeditV0	32.42361	0.03091	1	11	17	1	50
jeditV1	39.44761	0.03245	3	18	17	9	54
jeditV2	85.52344	0.21	4	27	9	1	58
log4j	7.04038	0.01	17	50	2	1	65
lucene	25.52924	0.01	14	24	2	1	52
poiV0	14.71215	0.02907	3	43	6	16	128
poiV1	59.65800	0.47390	15	9	16	19	134
synapse	45.56791	0.10599	2	39	11	1	132
velocity	37.99794	0.12500	5	39	9	1	142
xercesV0	31.96121	0.04975	14	50	18	4	132
xercesV1	20.14329	0.04677	24	40	8	9	132

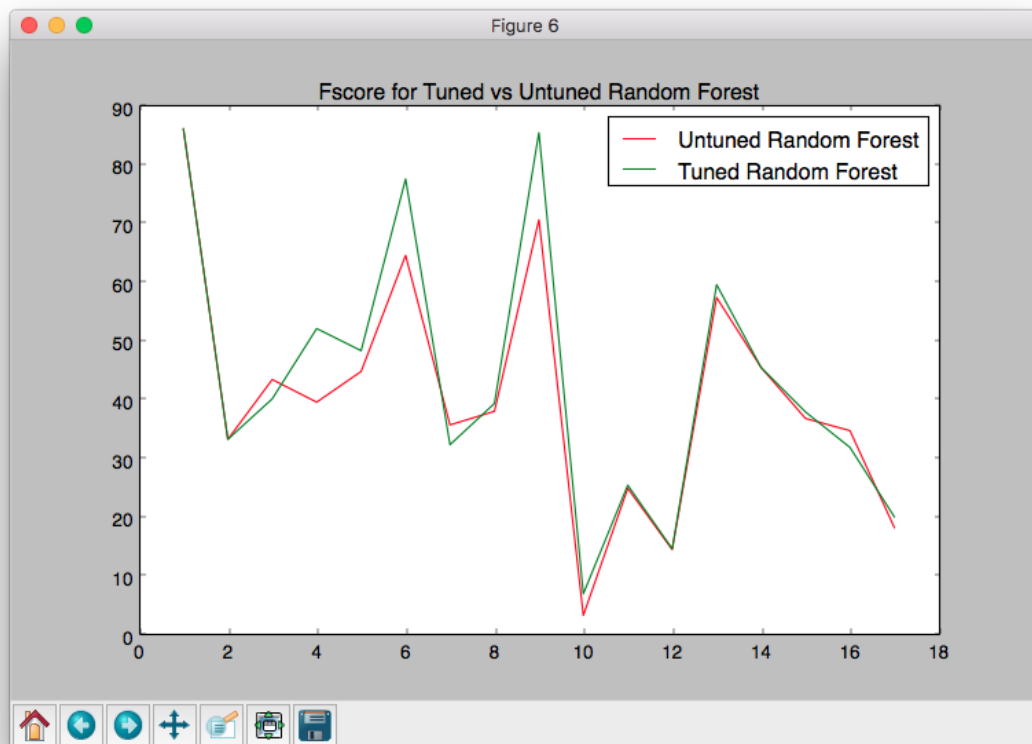


Figure 4.5: F-Measure for Tuned vs Untuned Random Forest

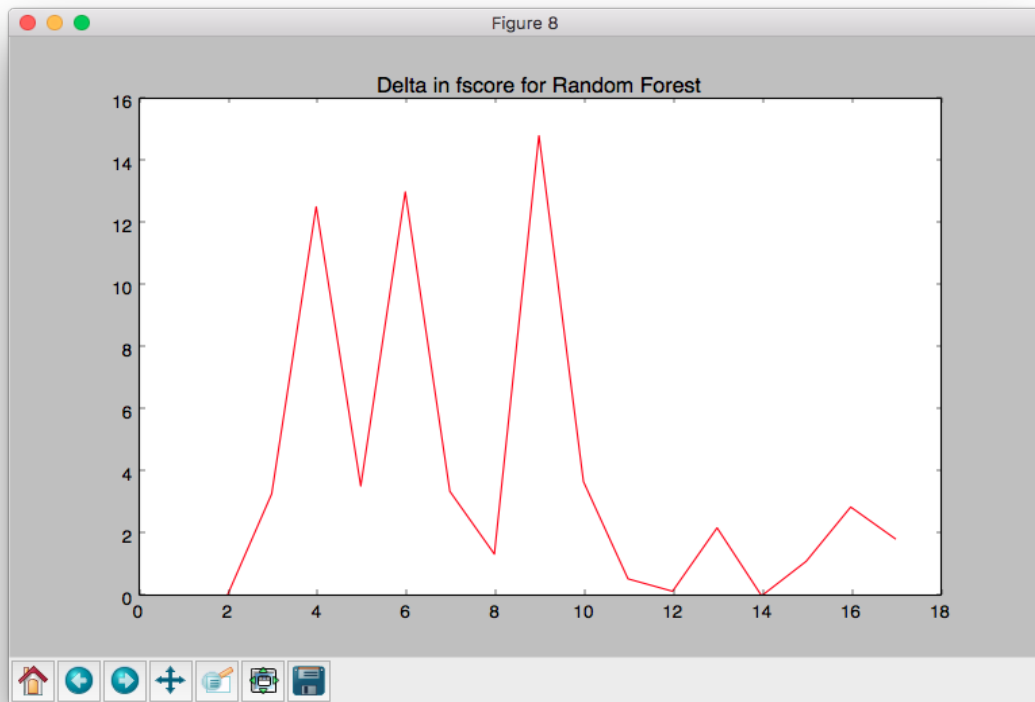


Figure 4.6: Delta in F-measure for Random Forest

C) CART USING PRECISION AS OPTIMISING GOAL

Dataset	Precision	Parameters				
		Threshold	max_feature	min_sample_split	min_samples_leaf	max_depth
antV0	87.16911	0.29195	15	6	1	30
antV1	25	0.77025	14	17	13	6
antV2	38.10515	0.77025	10	3	1	28
camelV0	23.97621	0.83379	15	7	12	4
camelV1	40.08975	0.09840	8	2	1	37
ivy	74.82758	0.10299	7	11	10	25
jeditV0	37.24989	0.21380	13	2	11	44
jeditV1	51.04166	0.13830	16	19	8	1
jeditV2	92.72727	0.08227	1	2	1	28
log4j	25.58944	0.12746	11	2	1	50
lucene	24.60080	0.28243	11	2	1	50
poiV0	39.64516	0.10013	14	2	14	12
poiV1	51.16576	0.05920	12	17	15	10
synapse	37.43078	0.12409	16	6	1	35
velocity	36.90497	0	17	2	1	7
xercesV0	30.58586	0.03072	17	8	6	42
xercesV1	42.66239	0.00051	1	2	5	1

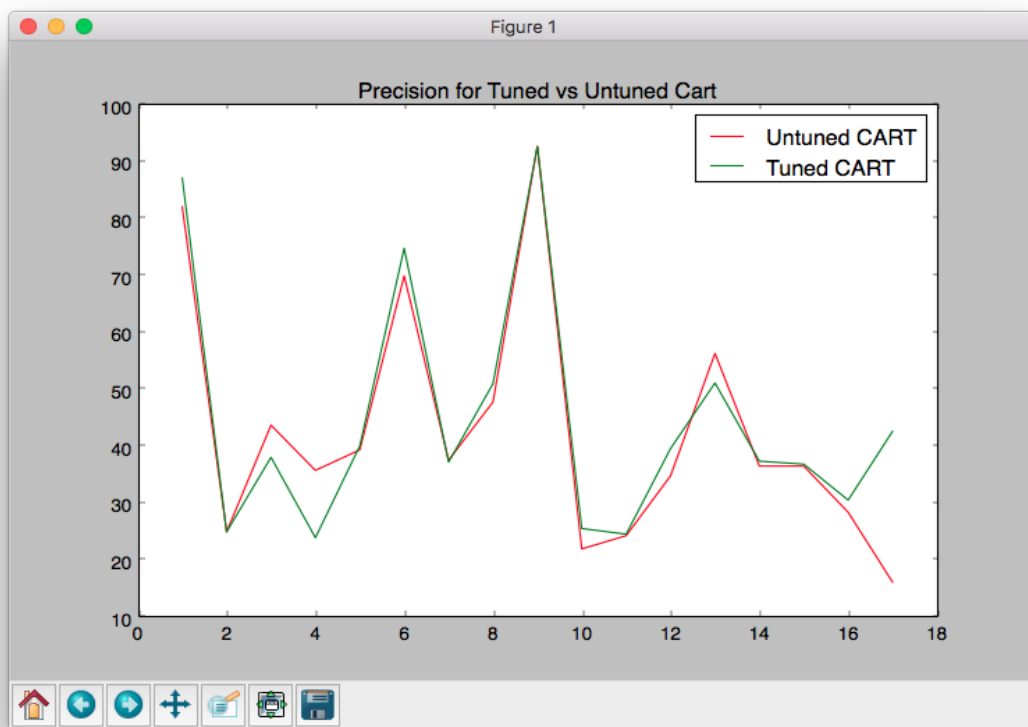


Figure 4.7: Precision for Tuned vs Untuned CART

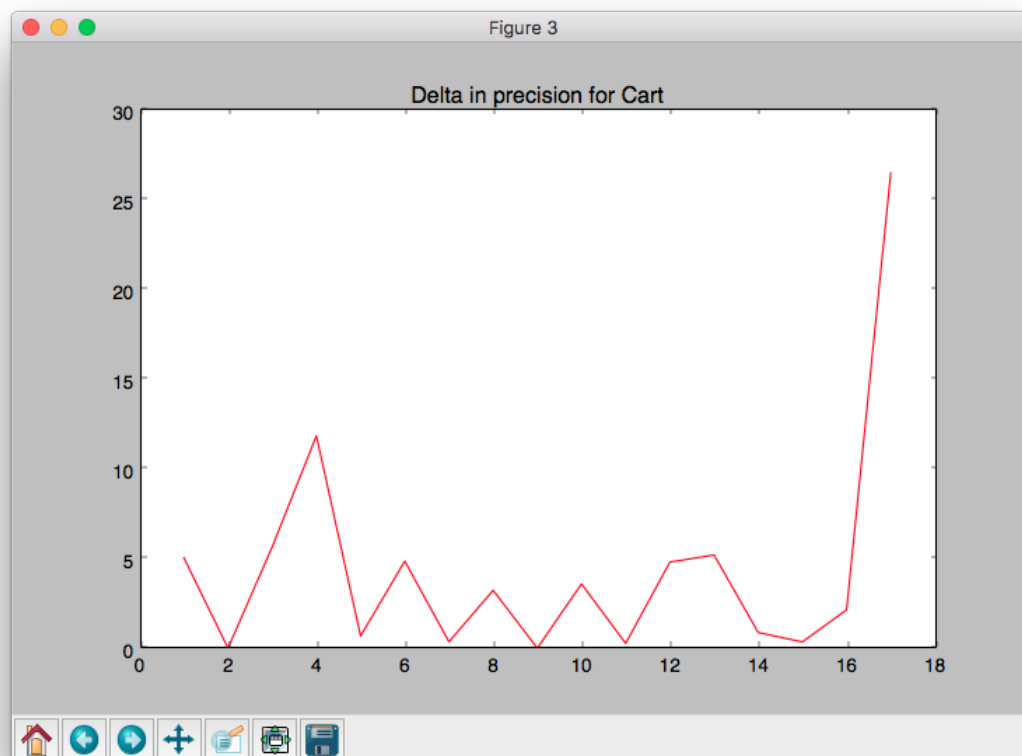


Figure 4.8: Delta in precision for CART

D) CART USING F-MEASURE AS OPTIMISING GOAL

Dataset	F-Measure	Parameters				
		Threshold	max_feature	min_sample_split	min_samples_leaf	max_depth
antV0	86.168032	0.46886	16	16	20	6
antV1	33.333333	0.04293	16	19	2	15
antV2	44.968650	0	10	9	2	13
camelV0	32.190293	0.71334	11	11	9	44
camelV1	43.674080	0.52299	4	13	1	27
ivy	57.647058	0.88934	1	11	15	1
jeditV0	34.844543	0.06356	13	2	16	18
jeditV1	76.171875	0.06549	6	10	20	26
jeditV2	86.58008	0.44666	1	3	20	50
log4j	16.15153	0	17	2	1	50
lucene	27.17951	0.43343	16	6	18	40
poiV0	18.07379	0.03188	1	14	3	1
poiV1	61.77111	0.04206	13	2	1	32
synapse	55.56791	0.97398	12	2	15	5
velocity	38.39882	0	17	2	1	8
xercesV0	32.60869	0.34557	17	2	1	50
xercesV1	22.54361	0	1	2	20	31

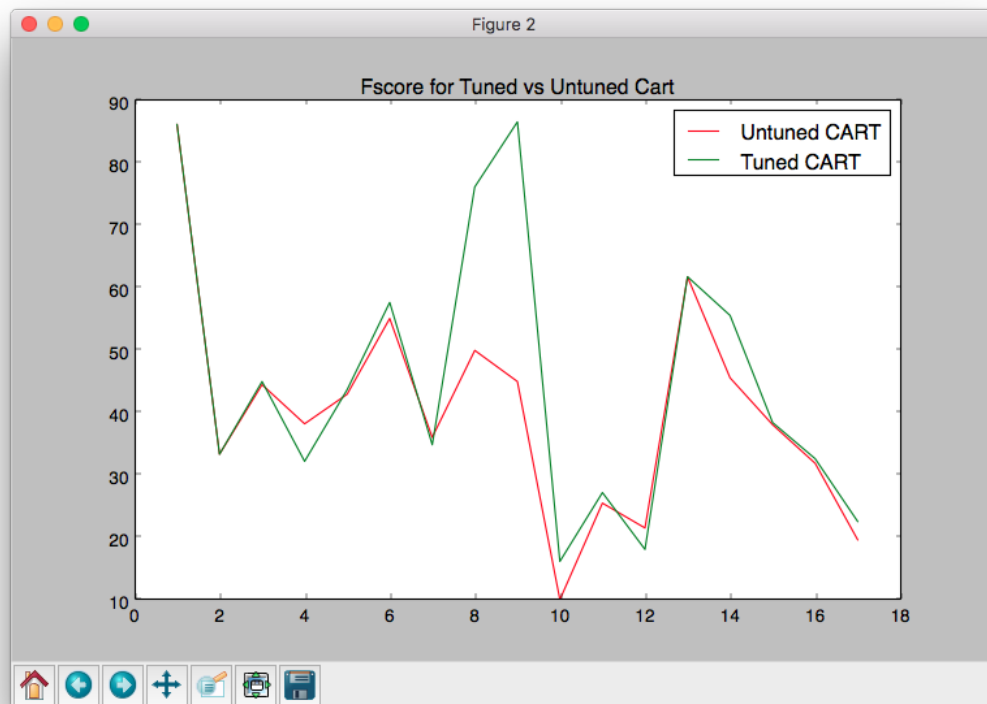


Figure 4.9: F-Measure for Tuned vs Untuned CART

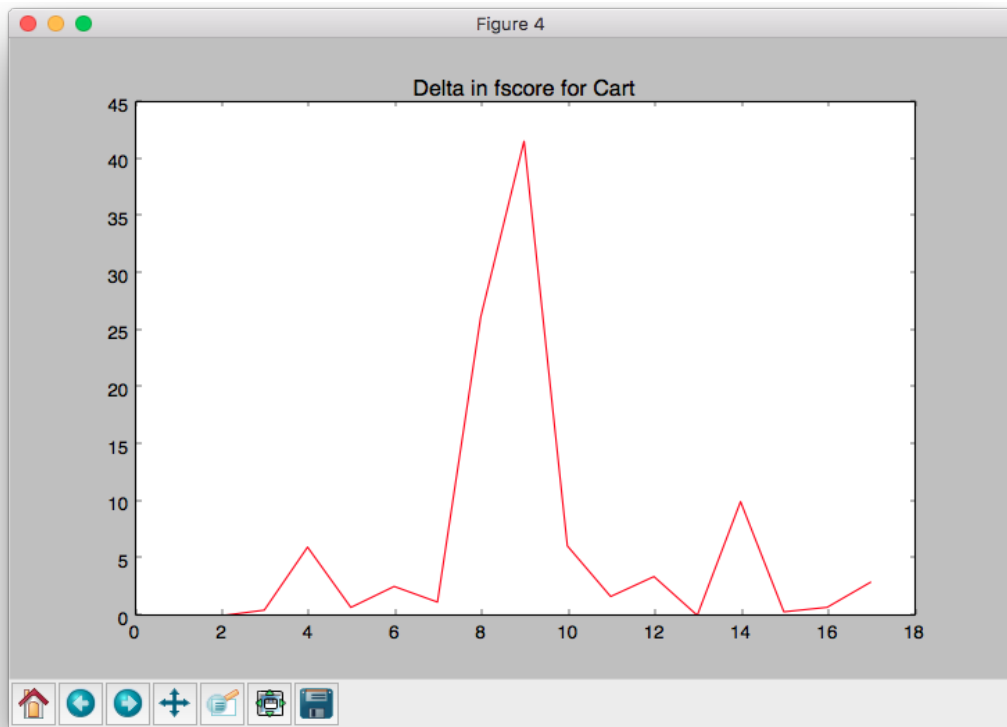


Figure 4.10: Delta in F-Measure for CART

Chapter 5 – Conclusion

I) Contributions

In this project, we studied the impact of Parameter Tuning in the field of Software Defect Prediction. Differential Evolution, a tuning algorithm was applied on machine learning algorithms like Random Forest and Classification And Regression Trees (CART) with the aim of optimising Precision or F-Measure. It was found that tuning parameters improved both these performance measures substantially as compared to default parameters.

It was also observed that tuning the parameters of the learning algorithms was simple and that Differential Evolution showed greater improvement as compared to CART.

Our exploration in the field of Parameter Tuning showed that when learning defect predictors for static code attributes^[4], analytics without tuning are considered harmful and misleading.

II) Future Work

It is now important to explore the implications of these conclusions to other kinds of software analytics. This project has investigated some learners using one optimizer. Hence, we can make no claim that DE is the best optimizer for all learners. Rather, our point is that there exists at least some learners whose performance can be dramatically improved by at least one simple optimization scheme.

Chapter 6 – References

- [1] T. Menzies, C. Pape, M. Rees-Jones, *The promise repository of empirical software engineering data*, 2015, URL: <http://openscience.us/repo>
- [2] D. Rodriguez, I. Herraiz, R. Harrison, *On software engineering repositories and their open problems*, in: Proceedings RAISE'12, 2012.
- [3] N. Nagappan, T. Ball, *Static analysis tools as early indicators of pre-release defect density*, in: ICSE '05, ACM, 2005, pp. 580–586.
- [4] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, *Benchmarking classification models for software defect prediction: a proposed framework and novel findings*, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.
- [5] R. Malhotra, *“Empirical Research in Software Engineering”*, CRC Press
- [6] Y. Singh and R. Malhotra, *“Object Oriented Software Engineering”*, PHI
- [7] R. Storn, K. Price, *Differential evolution—a simple and efficient heuristic for global optimization over continuous spaces*, J. Global Optim. 11 (4) (1997) 341–359.
- [8] L. Breiman, A. Cutler, *Random forests*, 2001, <https://www.stat.berkeley.edu/~breiman/RandomForests> .
- [9] J. Vesterstrom, R. Thomsen, *A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems*, IEEE Congress on Evolutionary Computation '04, 2004, doi: [10.1109/CEC.2004.1331139](https://doi.org/10.1109/CEC.2004.1331139) .
- [10] A.E. Hassan, *Predicting faults using the complexity of code changes*, in: *Proceedings of the 31st International Conference on Software Engineering*, in: ICSE '09, IEEE Computer Society, Washington, DC, USA, 2009, pp. 78–88, doi:[10.1109/ICSE.2009.5070510](https://doi.org/10.1109/ICSE.2009.5070510).