Major Project Report
ON

Comparison between Differential Evolution and Simulated Annealing for Parameter Tuning

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 major 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 Major 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.

DECLARATION

We hereby declare that the project work entitled "**Comparison between Differential Evolution and Simulated Annealing for Parameter Tuning**" submitted to CSE Department, DTU is a record of an original work done by us under the guidance of **Dr. Ruchika Malhotra**, Assistant Professor, CSE Department, DTU and this project work is submitted in the fulfilment of Major Project (7^{th} Semester). The results embodied in this thesis have not been submitted to any other University or Institute for the award of any degree or diploma.

Arjun Rajpal
2K14/SE/021

Dushyant Rathore
2K14/SE/029

Manav Middha
2K14/SE/043

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 Major project entitled "Comparison between Differential Evolution and Simulated Annealing for Parameter Tuning" 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

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 and Simulated Annealing as optimizers 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 at least 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 if 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 very 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] and Simulated Annealing [11] tuning methods. Impacts of both the methods on defect predictors have been investigated. It is shown that differential evolution is more effective than simulated annealing and other methods.

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.

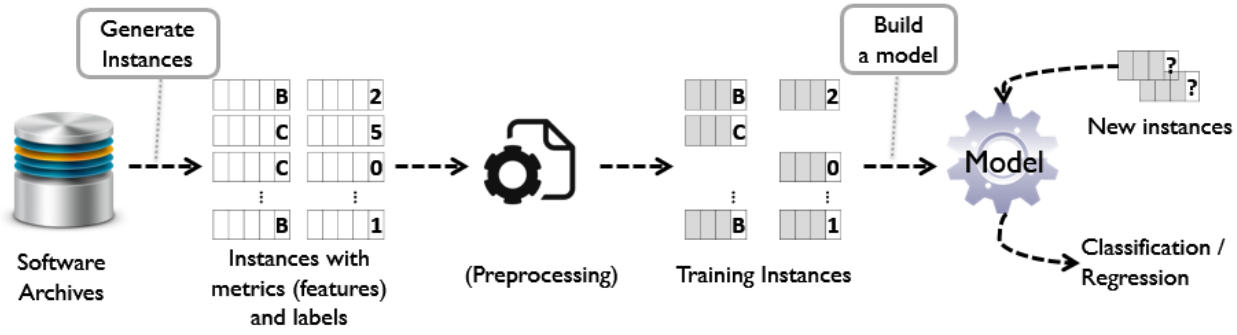

Figure 2.1: Defect Prediction using Machine Learning

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.

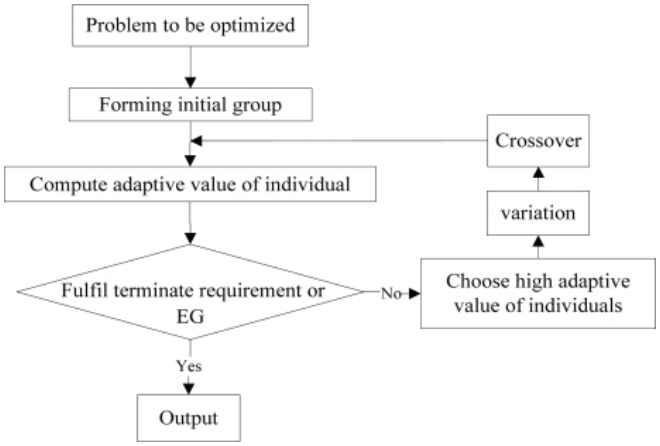

Figure 2.2: Differential Evolution flowchart

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 Simulated Annealing

Simulated Annealing[11] is a probabilistic technique for approximating the global optimum of a given function. This algorithm is often used when the search space is discrete and it is important to find the global optimum than a local maxima in limited time.

It is based on annealing in metallurgy where there is heating and controlled cooling of a material to increase crystal size and decrease defects. Heating and cooling affects the temperature and thermodynamic energy of the material.

This notion of slow cooling implemented in the Simulated Annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the global optimal solution. In general, the Simulated Annealing algorithms work as follows. At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and then decides to move to it or to stay with the current solution based on either one of two probabilities between which it chooses on the basis of the fact that the new solution is better or worse than the current one. During

the search, the temperature is progressively decreased from an initial positive value to zero and affects the two probabilities: at each step, the probability of moving to a better new solution is either kept to 1 or is changed towards a positive value; instead, the probability of moving to a worse new solution is progressively changed towards zero.

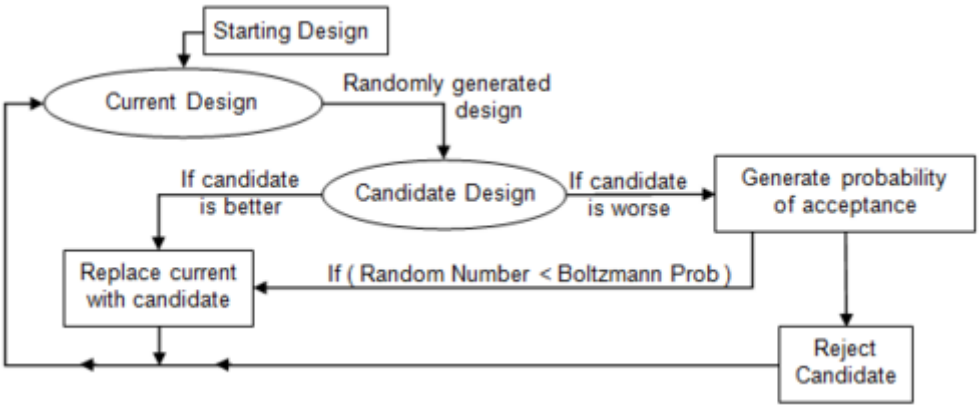

Figure 2.3: Simulated Annealing flowchart

2.4 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.

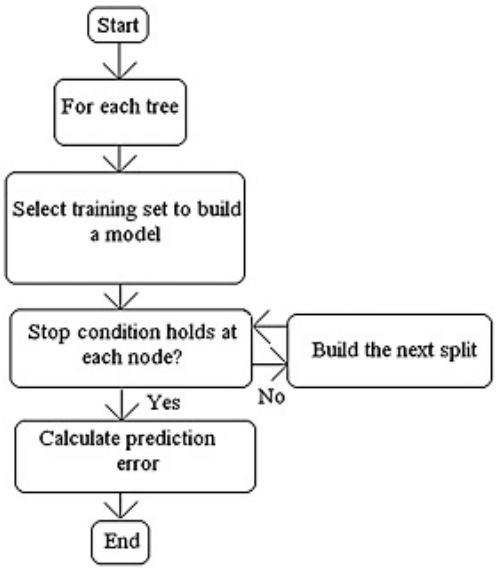

Figure 2.4: Random Forest flowchart

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.

2.5 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.

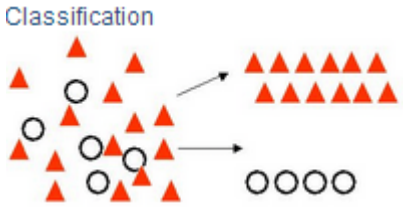

Figure 2.5: Classification Trees

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

Figure 2.6: 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.

2.6 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.

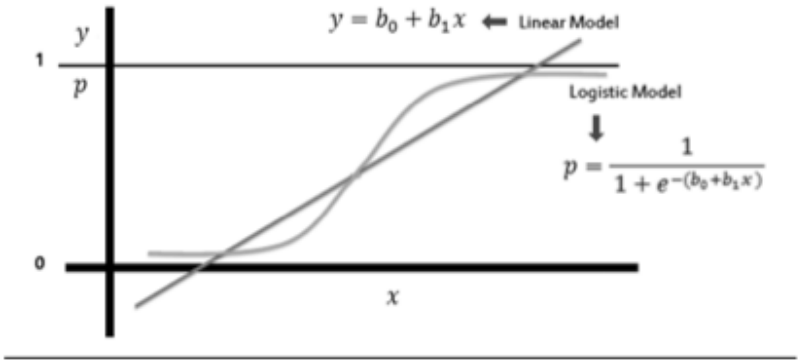

Figure 2.7: Comparison of linear regression model and logistic regression model

In the logistic regression the constant (b_o) 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.

$$\frac{p}{1-p} = \exp(b_0 + b_1 x)$$

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

$$\ln\left(\frac{p}{1-p}\right) = b_0 + b_1 x$$

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.

$$p = \frac{1}{1 + e^{-(b_0 + b_1 x_1 + x_2 b_2 + \cdots + b_p x_p)}}$$

2.7 Wilcoxon Signed Rank Test

The Wilcoxon signed-rank test[12] is a non-parametric statistical hypothesis test used when comparing two related samples, matched samples, or repeated measurements on a single sample to assess whether their population mean ranks differ (i.e. it is a paired difference test).

Algorithm:

Let N be the sample size i.e. the number of pairs. Thus there are a total of 2N data points.

1. For pairs $i = 1, \ldots, N$, calculate $|x_{2,i} - x_{1,i}|$ and $sgn(x_{2,i} - x_{1,i})$ where sgn is sign function.
2. Exclude pairs with $|x_{2,i} - x_{1,i}| = 0$. Let N_r be the reduced sample size.

3. Order the remaining N_r pairs from the smallest absolute difference to largest absolute difference$|x_{2,i} - x_{1,i}|$.
4. Rank the pairs, starting with the smallest as 1. Ties receive a rank equal to the average of the ranks they span. Let R_i denote the rank.
5. Calculate the test static W

$$W = \sum_{i=1}^{N_r} [sgn(x_{2,i} - x_{1,i}) * R_i]$$

6. Under null hypothesis, W follows a specific distribution with no simple expression. This distribution has an expected value of 0 and a variance of $\frac{N_r(N_r+1)(2N_r+1)}{6}$

   W can be compared to a critical value from a reference table.
   The two-sided test consists in rejecting H_0 if $|W| > W_{critical,N_r}$.
7. As N_r increases, the sampling distribution of W converges to a normal distribution.

   For $N_r \geq 10$,a z_score can be calculated as $z = \frac{W}{\sigma_W}, \sigma_W = \sqrt{\frac{N_r(N_r+1)(2N_r+1)}{6}}$
   To perform a two-sided test ,reject H_0 if $|Z| \geq Z_{critical}$.

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 and simulated annealing for defect prediction, we performed the following study.

3.1 Tuning Parameters:

Our DE and simulated annealing techniques explored the parameter space of Figure 3.1. Specifically, since Tantithamthavorn et al. divide each tuning range into 5 bins (if applicable), we also used 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 simulated annealing 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 simulated annealing 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 denore defects I 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) **∑Xi^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 | =>8105 R2 -> | 7 | 9 | -50 | =>2630 R3 -> | 4 | 2 | -9.2| =>104.64 R4 -> | 8.5 | 7 | 9 | =>202.25

objective function value is calculated using the given objective function. In this case, it's **∑Xi^2**. So for R1, objective function value will be **-90^2+2^2+2^2 = 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,

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:

MV = | 13.25 | 13.5 | -50.1 | =>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).

Target-> |-90| 2| 1| Mutant-> | 13.25 | 13.5 | -50.1 |

random num - 0.21, => `Child -> |13.25| -- | -- |` random num - 0.97, => `Child -> |13.25| 2 | -- |` random num - 0.80, => `Child -> |13.25| 2 | -50.1 |` Trial vector/child vector -> | 13.25 | 2 | -50.1 | =>2689.57

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 -> |-90 | 2 | 1 | =>8105 Trial vector/child vector -> | 13.25 | 2 | -50.1 | =>2689.57

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

G1 G2 G3 objective fn value R1 -> | 13.25 | 2 | -50.1 | =>2689.57 R2 -> | 7 | 9 | -50 | =>2500 R3 -> | 4 | 2 | -9.2 | =>104.64 R4 -> | -8.5 | 7 | 9 | =>202.25

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

Simulated Annealing

1) Initialize the system configuration.
   Randomize **x(0)**.

2) Initialise T with a large value.

3) **Repeat:**
   a) **Repeat:**
      i) Apply random perturbations to the state **x = x + Δx.**
      ii) Evaluate **ΔE(x) = E(x + Δx) − E(x)**:
      if ΔE(x) < 0, keep the new state;
      otherwise, accept the new state with probability $P = e^{-(\Delta E/T)}$.

      until the number of accepted transitions is below a threshold level.

   b) Set T = T − ΔT .

   until T is small enough

3.4 Basic Overview of Codes:

1. 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)

        oldPopulation = []
        globalPopulation = []

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

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

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

        s_Best = getBestSolution(global_best)

    return s_Best
```

2. Simulated Annealing

```
def SA(no_of_iterations,T,T_min,alpha,no_of_parameters,dataset):

    solution = initialiseSolution(no_of_parameters)
    solution['cost'] = score(solution,dataset)
    count=0

    while T>T_min:

        count += 1
        i=1
        while i<= no_of_iterations:
            new_solution = neighbour(solution['tunings'],no_of_parameters)
            # new_solution =
fast_schedule(solution['tunings'],no_of_parameters,T)
            # new_solution = cauchy_schedule(solution['tunings'],T)

            new_solution['cost'] = score(new_solution,dataset)

            if new_solution['cost']>solution['cost']:
                solution = new_solution
            else:
                ap =
acceptance_probability(solution['cost'],new_solution['cost'],T)
                if ap>np.random.uniform(0,1):
                    solution = new_solution

            i = i+1

        print "Iteration at Temp" + str(T)
        T = T*alpha
        # T = update_temp_fast_schedule(count,T)
        # T = update_temp_cauchy_schedule(count,T)

    return solution
```

3. Modified Random Forest

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

```
    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)

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

    return p
```

4. CART

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

    ct = DecisionTreeClassifier(max_depth=tunings[4],
min_samples_split=tunings[2], min_samples_leaf=tunings[3],
                                max_features=tunings[1],
min_impurity_split=tunings[0])
    ct.fit(a, b)
    pred = ct.predict(c)
    p = precision_score(d, pred, average='weighted')

    return p
```

5. Wilcoxon Signed Rank Test

```
def signed_rank(before_tuning, after_tuning):

    l = []

    for i in range(len(before_tuning)):
            d={}
            d['before_tuning'] = before_tuning[i]
            d['after_tuning'] = after_tuning[i]
            d['difference'] = after_tuning[i]-before_tuning[i]
            d['absolute_value'] = abs(d['difference'])

            if d['difference']>=0:
                    d['sign'] = '+'
            else:
                    d['sign'] = '-'

            l.append(d)

    l = sorted(l, key=lambda k: k['absolute_value'])
```

```
for i in range(len(l)):
        l[i]['rank'] = i+1

for i in range(len(l)):
        print l[i]['difference']

CRITICAL_VALUE = 41

sum_pos = 0
sum_neg = 0

for i in range(len(l)):
        if l[i]['sign']=='+':
                sum_pos += l[i]['rank']
        else:
                sum_neg += l[i]['rank']


min_sum = min(sum_pos,sum_neg)

if min_sum < CRITICAL_VALUE:
        return "Null hypothesis rejected, thus they are different"
else:
        return "Null hypothesis is accepted, thus they are same"
```

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

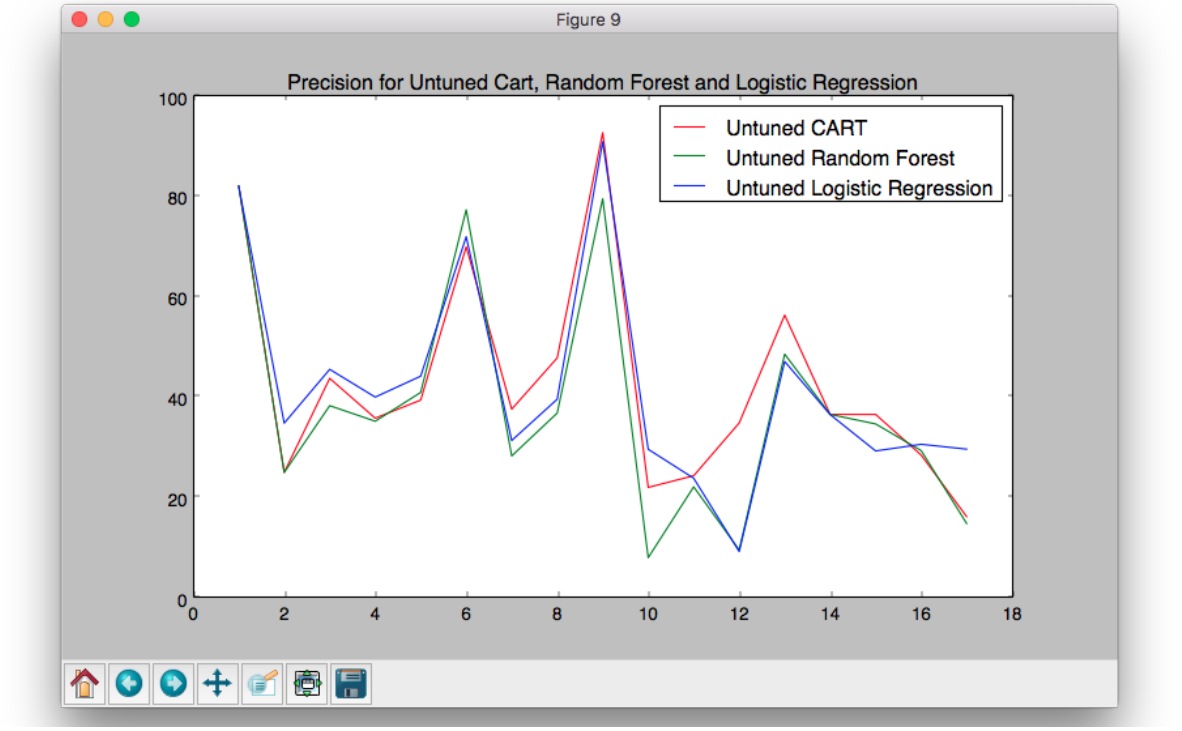

Figure 4.1: Precision for Untuned CART, Random Forest & Logistics Regression

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

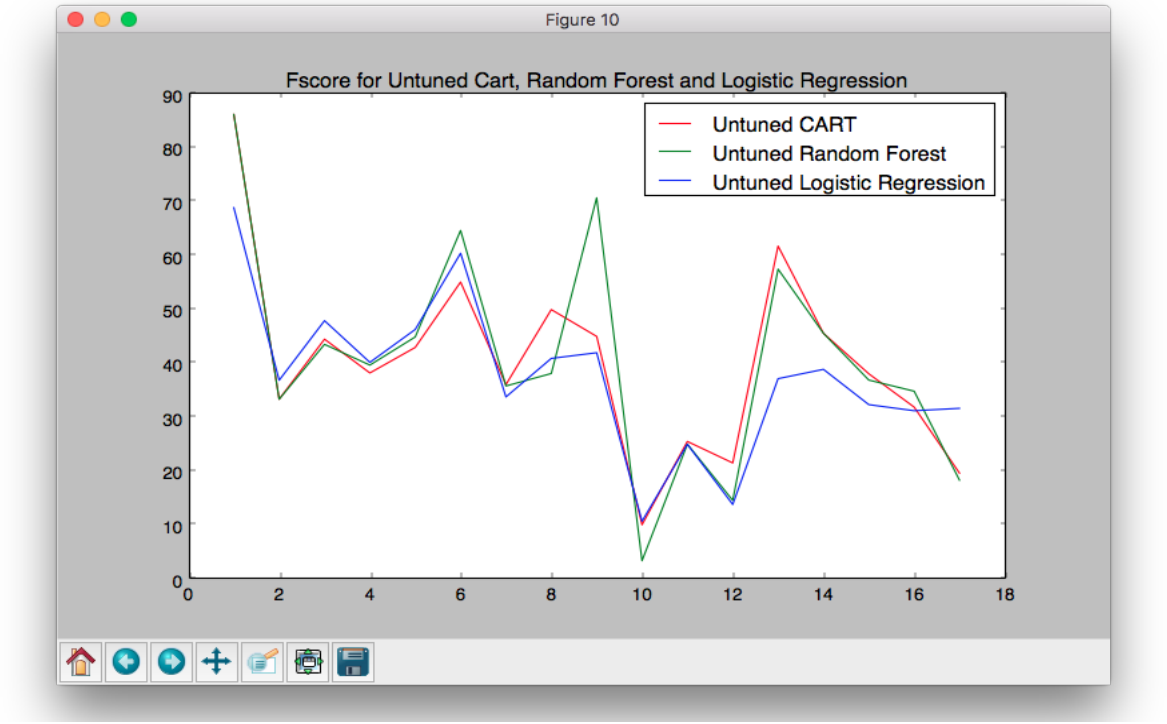

Figure 4.2: F-measure for Untuned CART, Random Forest & Logistics Regression

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 WITH DIFFERENTIAL EVOLUTION

Datasets	Precision	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ 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

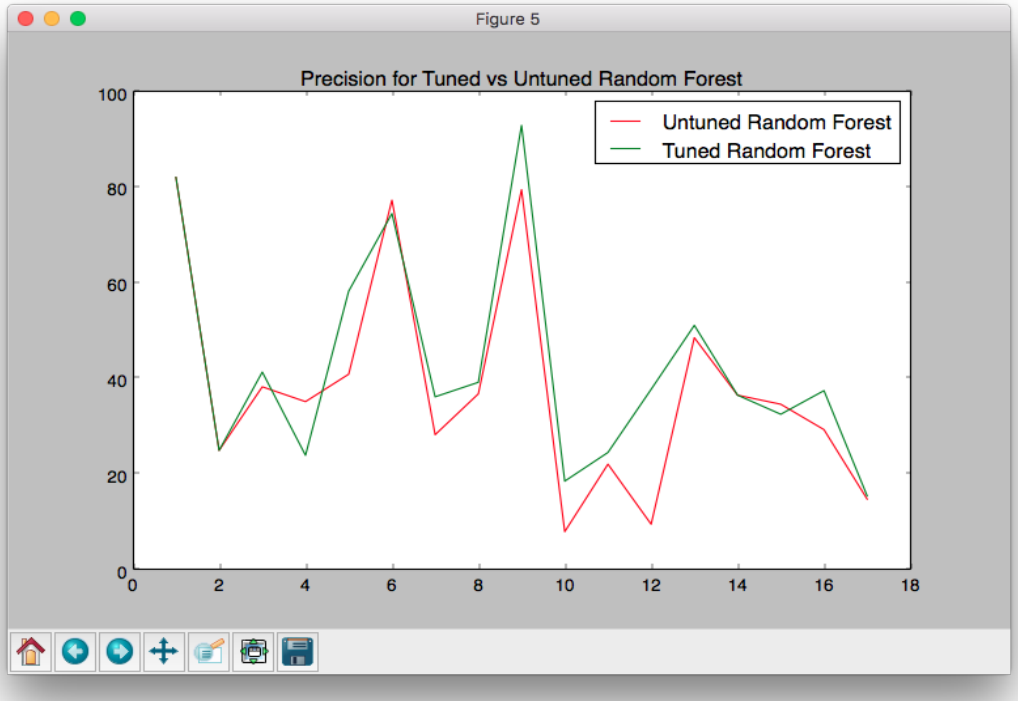

Figure 4.3: Precision for Tuned vs Untuned Random Forest

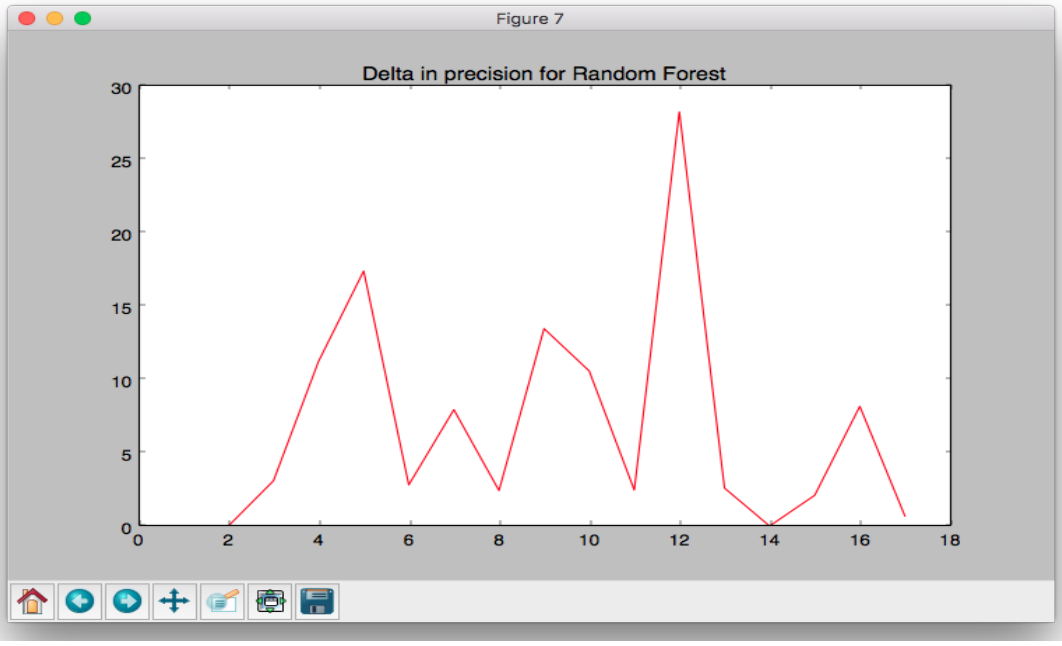

Figure 4.4: Delta in Precision for Random Forest

B) RANDOM FOREST USING F-MEASURE AS OPTIMISING GOAL WITH DIFFERENTIAL EVOLUTION

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

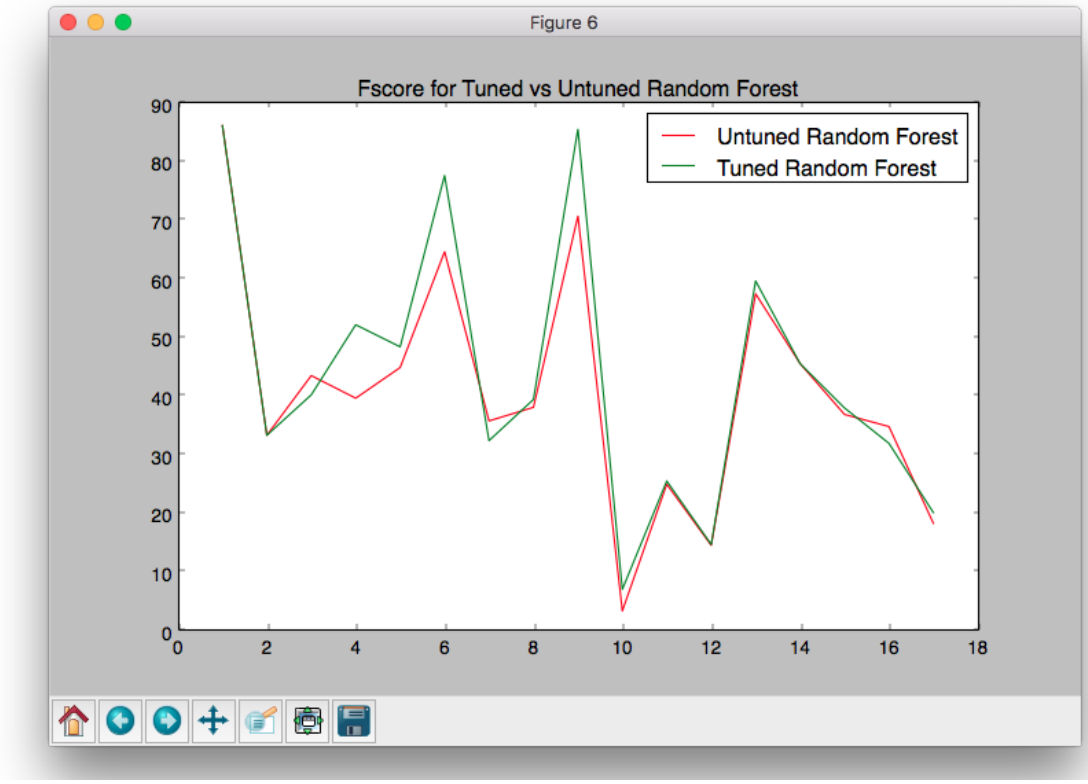

Figure 4.5: F-Measure for Tuned vs Untuned Random Forest

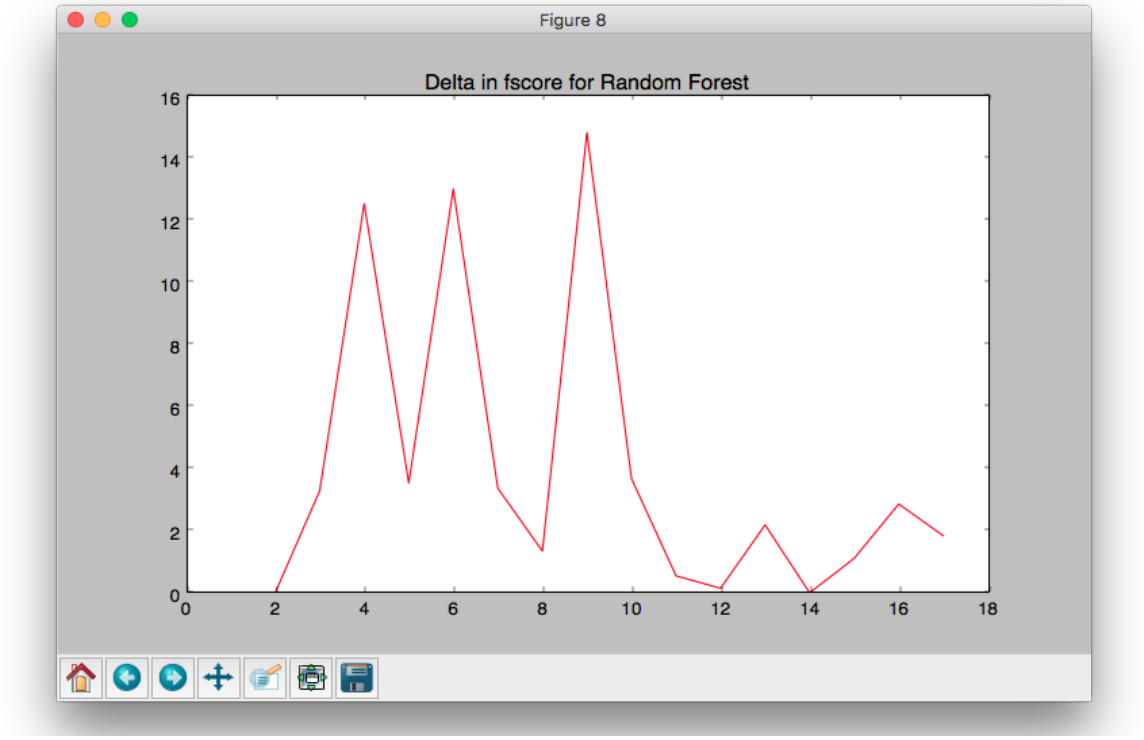

Figure 4.6: Delta in F-measure for Random Forest

C) CART USING PRECISION AS OPTIMISING GOAL WITH DIFFERENTIAL EVOLUTION

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

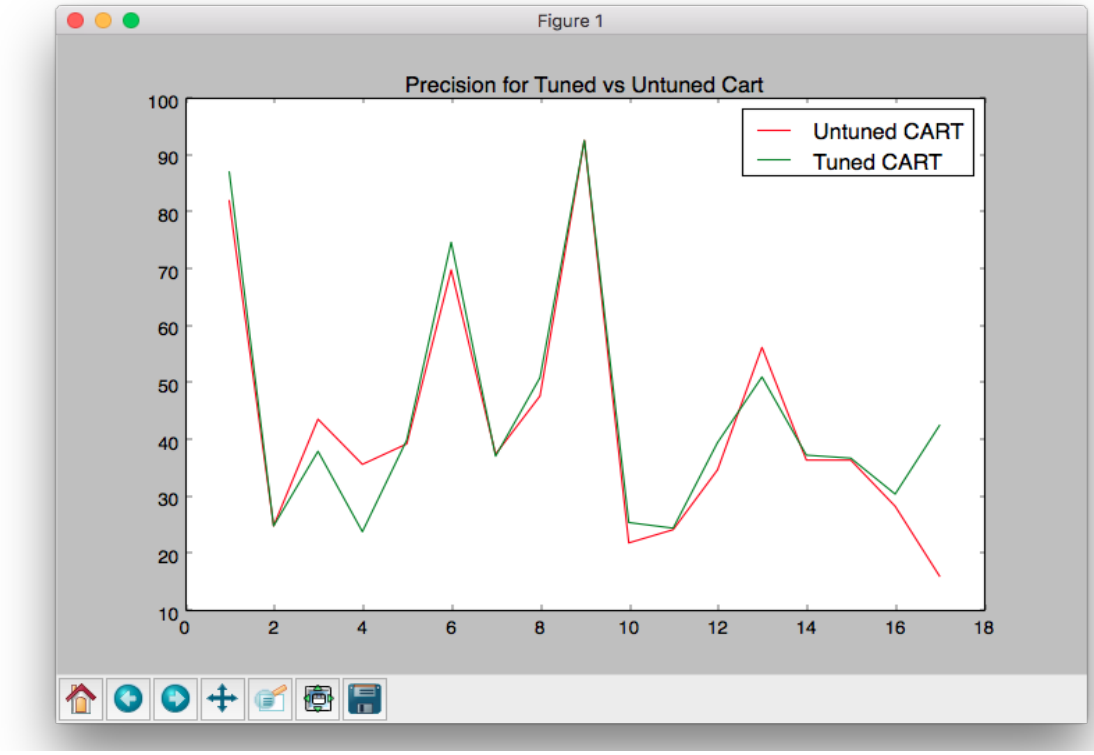

Figure 4.7: Precision for Tuned vs Untuned CART

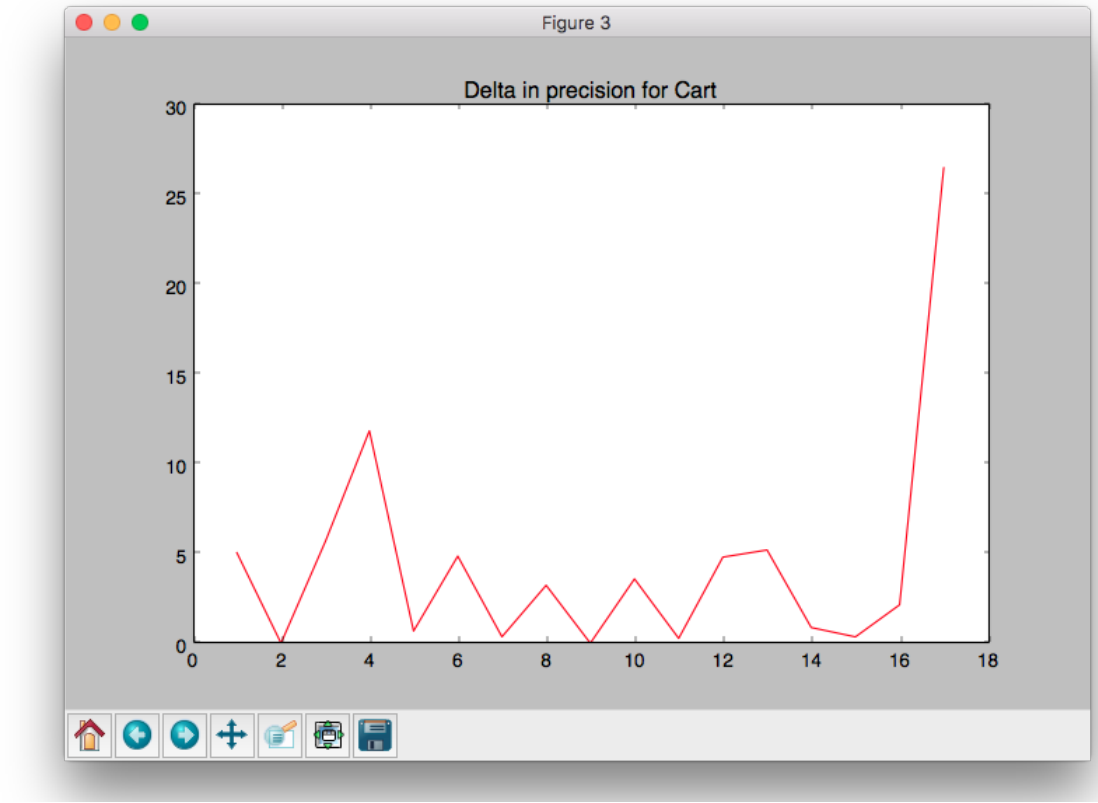

Figure 4.8: Delta in precision for CART

D) CART USING F-MEASURE AS OPTIMISING GOAL WITH DIFFERENTIAL EVOLUTION

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

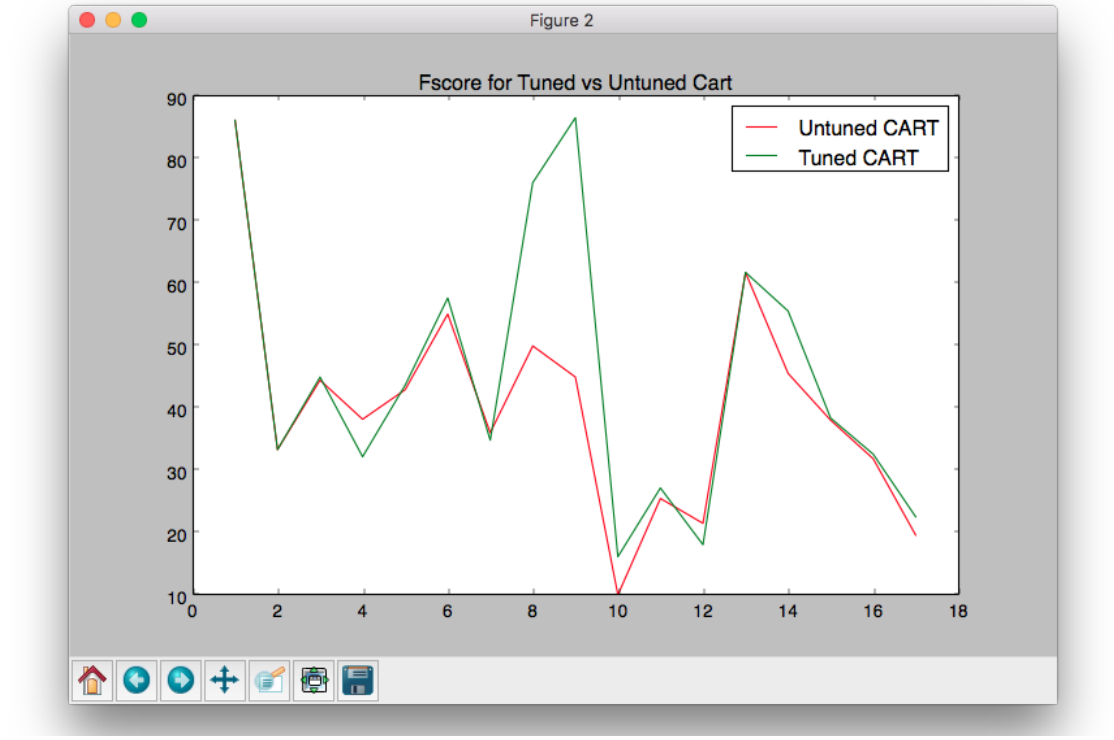

Figure 4.9: F-Measure for Tuned vs Untuned CART

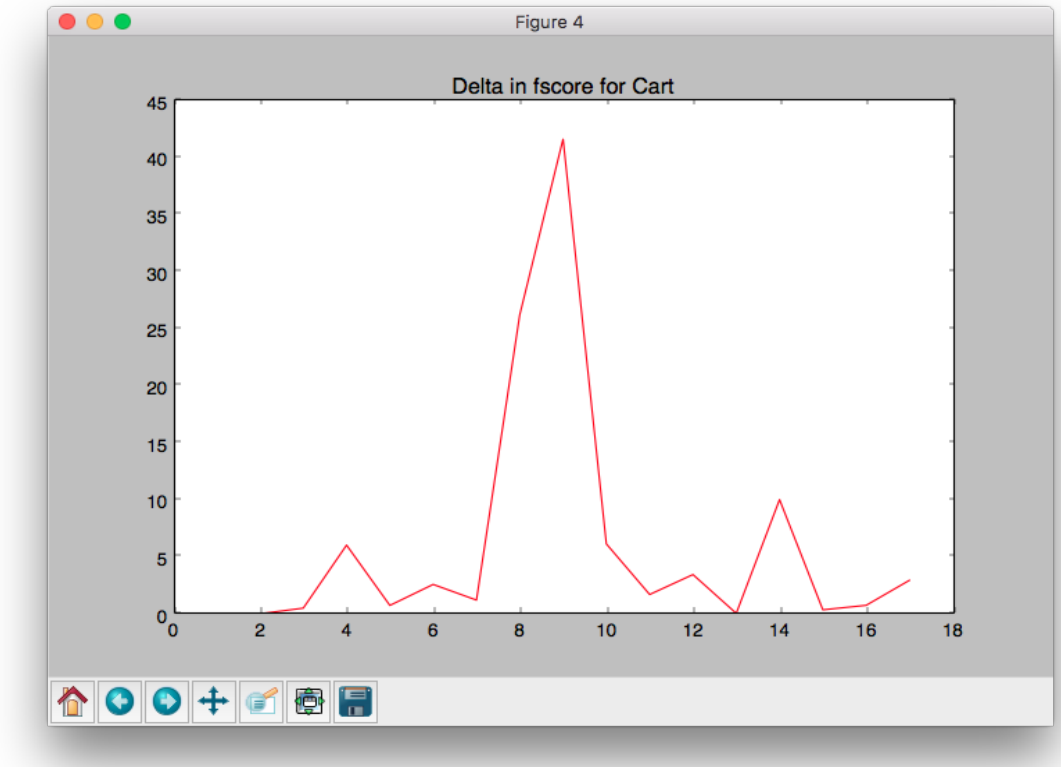

Figure 4.10: Delta in F-Measure for CART

E) RANDOM FOREST USING PRECISION AS OPTIMISING GOAL WITH SIMULATED ANNEALING

E.1) NEIGHBOUR SELECTION

Datasets	Precision	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	82.12890625	1	13	2	16	18	122
antV1	25.0	1	1	34	9	3	115
antV2	31.3869937582	1	2	2	18	14	84
camelV0	23.9762187872	1	19	37	2	10	136
camelV1	28.8620416478	1	7	10	6	3	117
ivy	49.0	1	13	40	13	8	79
jeditV0	21.9355872456	1	10	49	6	5	136
jeditV1	31.640625	1	14	2	11	11	89
jeditV2	82.6446280992	1	12	45	3	11	148
log4j	0.907029478458	1	5	29	8	5	52
lucene	11.5533014633	1	2	11	6	6	112
poiV0	9.64002341311	1	1	47	4	17	88
poiV1	51.1657653778	1	12	21	4	11	140
synapse	36.5603028664	1	14	14	12	8	92
velocity	19.0006574622	1	1	35	17	18	105
xercesV0	10.165931527	1	7	30	12	18	118
xercesV1	8.71397975588	1	7	47	7	16	122

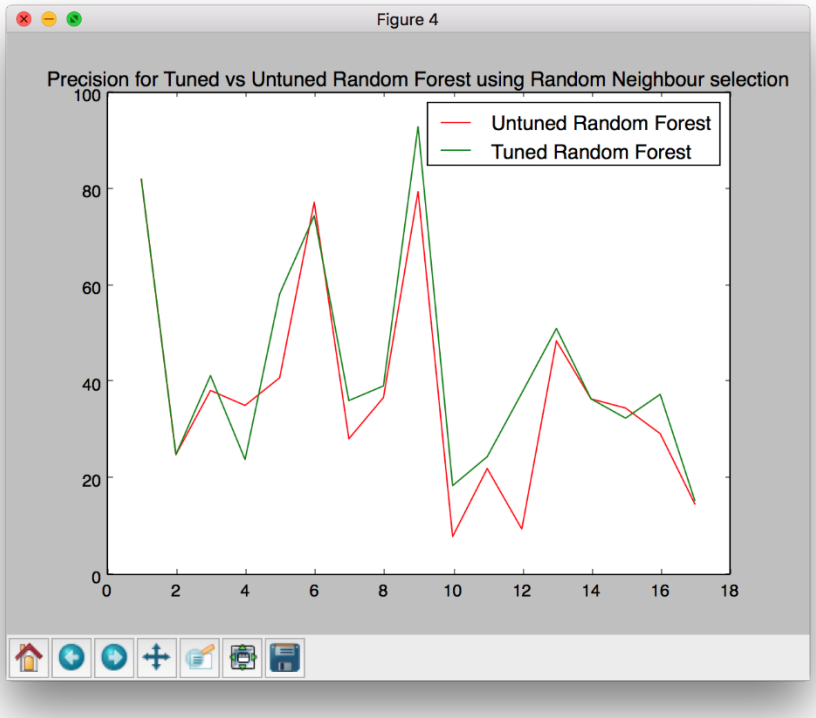

Figure 4.11: Precision for Tuned vs Untuned Random Forest using Random Neighbour Selection

E.2) FAST SCHEDULING

Datasets	Precision	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	82.12890625	1	11	7	17	1	141
antV1	25.0	1	1	46	5	14	145
antV2	31.3869937582	0.66	16	10	2	13	104
camelV0	23.9762187872	1	1	20	2	12	50
camelV1	28.8620416478	0.9187	1	2	2	17	50
ivy	49.0	1	13	8	20	14	185
jeditV0	21.9355872456	0.8515	1	15	16	20	99
jeditV1	37.8566	0.2396	4	49	2	1	109
jeditV2	72.7272	0.01	12	18	9	1	60
log4j	0.9070	0.6298	1	49	11	12	128
lucene	11.5533	1	1	9	12	11	52
poiV0	9.6400	1	1	6	16	14	51
poiV1	51.1657	0.0619	1	5	2	1	77
synapse	36.5603	0.0518	18	33	2	7	150
velocity	19.00	0.9069	13	43	7	1	50
xercesV0	10.1659	0.0133	1	45	2	16	74
xercesV1	8.7139	0.6122	17	2	15	1	50

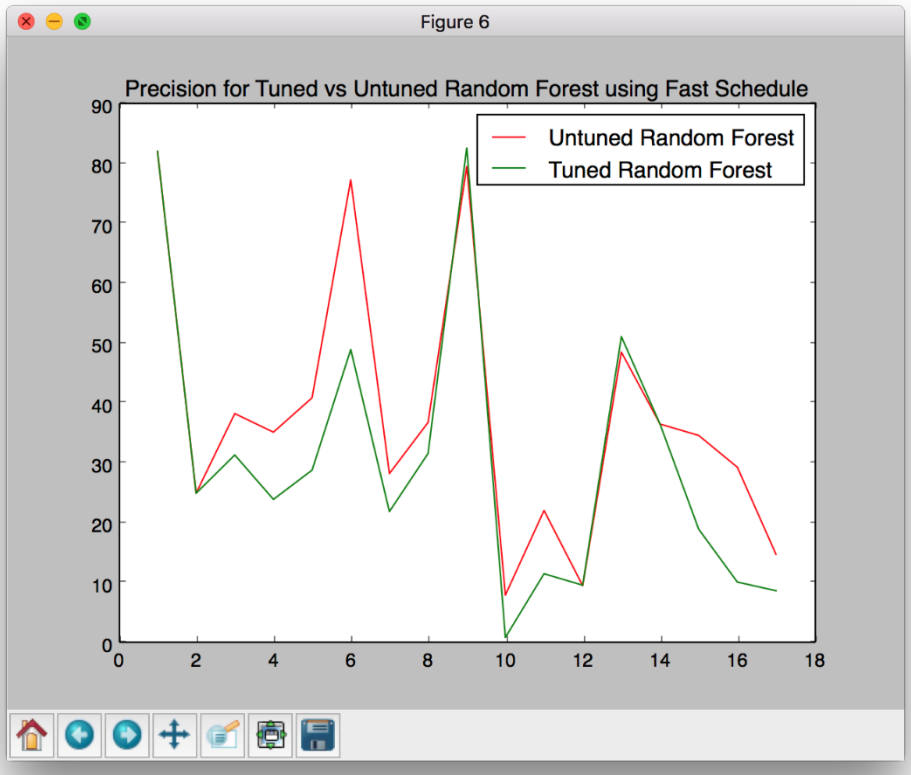

Figure 4.12: Precision for Tuned vs Untuned Random Forest using Fast Schedule

E.3) CAUCHY SCHEDULIING

Datasets	Precision	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	82.1289	0.782	1	2	2	1	50
antV1	25.0	0.3765	1	2	2	1	50
antV2	31.3869	0.9580	1	2	2	1	50
camelV0	23.9762	0.3798	1	2	2	1	50
camelV1	28.8620	0.0403	1	2	2	1	50
ivy	61.9948	0.5500	1	2	2	1	50
jeditV0	21.9355	0.5434	1	2	2	1	50
jeditV1	31.6406	0.3952	1	2	2	1	50
jeditV2	82.6446	0.9876	1	2	2	1	50
log4j	0.9070	0.0128	1	2	2	1	50
lucene	11.5533	0.0748	1	2	2	1	50
poiV0	9.6400	0.03418	1	2	2	1	50
poiV1	51.1657	0.7582	1	2	2	1	50
synapse	36.5603	0.8913	1	2	2	1	50
velocity	19.0006	0.0104	1	2	2	1	50
xercesV0	10.1659	0.9676	1	2	2	1	50
xercesV1	17.3464	0.2271	1	2	2	1	50

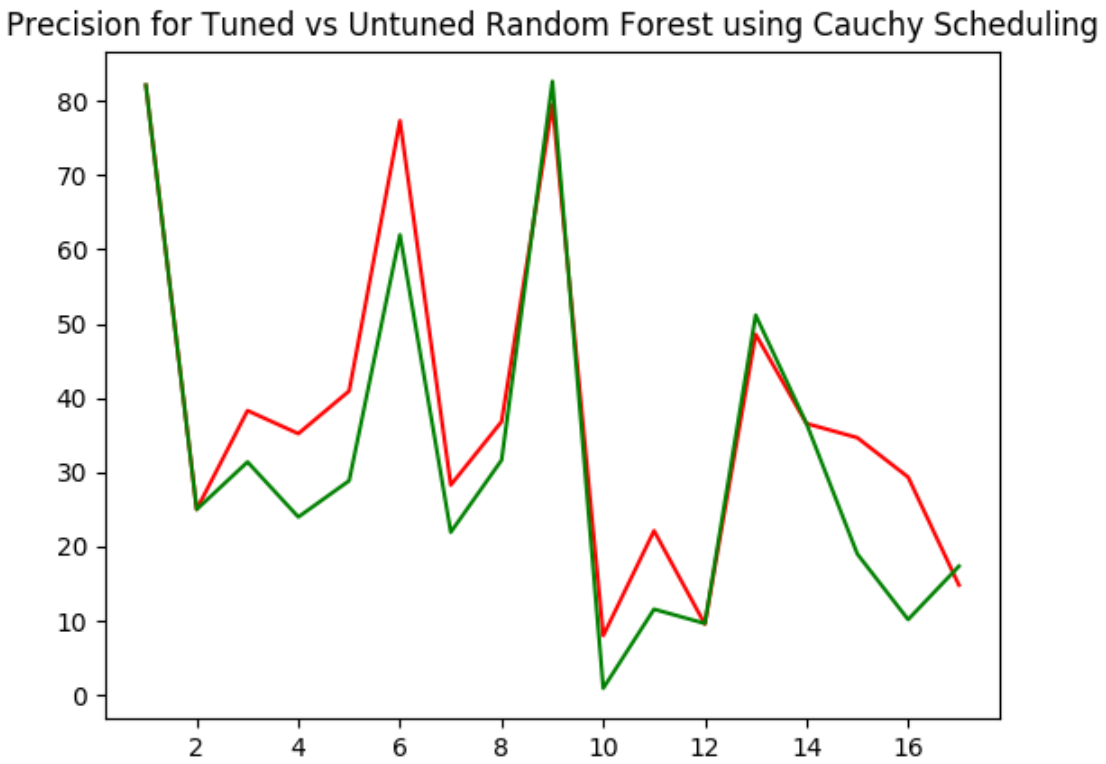

Figure 4.13: Precision for Tuned vs Untuned Random Forest using Cauchy Schedule

F) RANDOM FOREST USING F-SCORE AS OPTIMISING GOAL WITH SIMULATED ANNEALING

F.1) NEIGHBOUR SELECTION

Datasets	F-SCORE	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	86.168	1	3	46	2	17	94
antV1	33.33	1	18	4	6	4	103
antV2	40.2335	1	18	43	18	10	86
camelV0	32.1902	1	6	17	15	7	133
camelV1	37.5506	1	14	20	13	4	77
ivy	57.6470	1	15	21	13	2	140
jeditV0	29.8777	1	11	35	9	13	98
jeditV1	40.5	1	11	6	16	15	88
jeditV2	86.5800	1	4	22	12	10	68
log4j	1.6563	1	19	33	15	9	138
lucene	17.2450	1	13	31	10	11	83
poiV0	14.7121	1	9	27	11	9	57
poiV1	59.6580	1	9	20	5	5	119
synapse	45.5679	1	12	48	9	1	134
velocity	26.4652	1	9	3	14	3	99
xercesV0	15.4164	1	4	36	16	1	128
xercesV1	13.4558	1	2	6	3	13	111

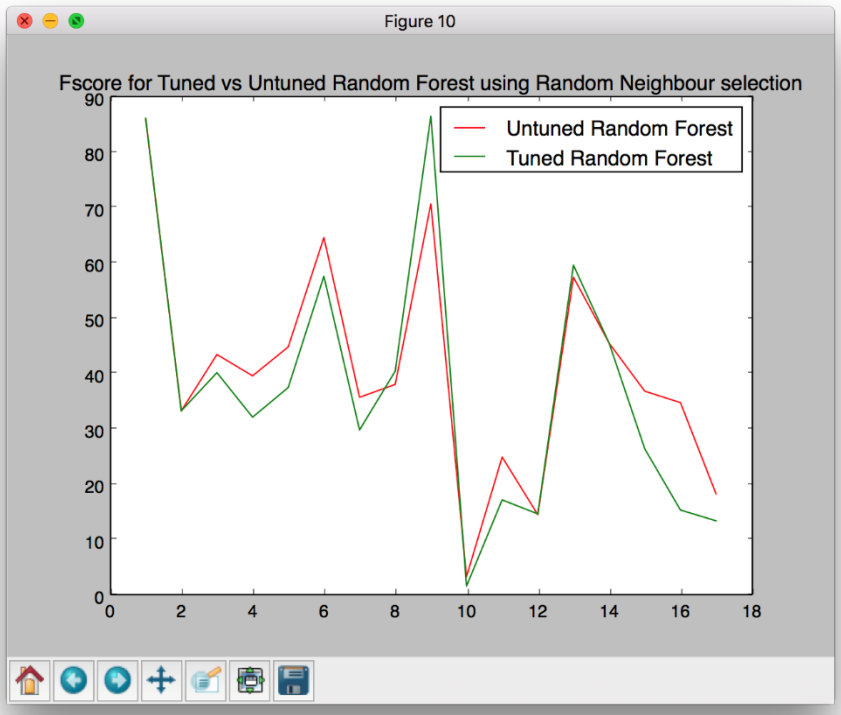

Figure 4.14: F-Score for Tuned vs Untuned Random Forest using Neighbour Selection

F.2) FAST SCHEDULING

Datasets	F-SCORE	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	86.1680	0.4135	9	49	5	18	94
antV1	33.33	1	10	3	3	19	50
antV2	40.2335	1	8	30	3	8	10
camelV0	32.1902	0.0189	20	2	5	1	94
camelV1	39.7373	0.01	1	44	2	1	138
ivy	57.6470	0.9918	2	2	7	20	76
jeditV0	30.4019	0.6894	1	16	7	9	50
jeditV1	45.6789	0.01	9	2	6	4	145
jeditV2	86.5800	0.998	20	46	19	1	59
log4j	1.6563	0.988	4	3	3	1	50
lucene	17.2450	0.01	1	2	16	3	149
poiV0	14.7121	0.1485	1	42	4	9	58
poiV1	59.6580	0.3303	1	8	15	1	116
synapse	45.5679	0.6037	13	49	14	6	58
velocity	26.4652	0.8470	20	12	2	1	147
xercesV0	15.4164	1	11	14	2	15	149
xercesV1	19.6386	0.1867	2	20	2	1	52

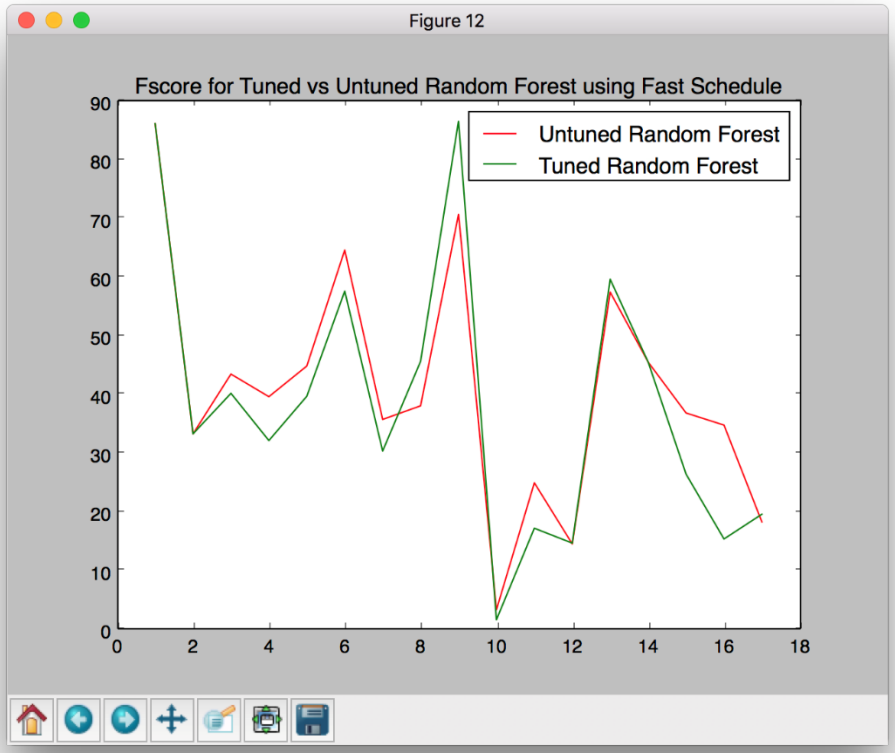

Figure 4.15: F-Score for Tuned vs Untuned Random Forest using Fast Schedule

F.3) CAUCHY SCHEDULIING

Datasets	F-SCORE	Parameters					
		Threshold	max_ feature	max_ leaf_ nodes	min_ sample_ split	min_ samples_ leaf	n_estimators
antV0	86.1680	0.4958	1	2	2	1	50
antV1	33.33	0.0350	1	2	2	1	50
antV2	40.2335	0.01	1	2	2	1	50
camelV0	32.1902	0.0307	1	2	2	1	50
camelV1	37.5506	0.9994	1	2	2	1	50
ivy	57.6470	0.9913	1	2	2	1	50
jeditV0	29.8777	0.77618	1	2	2	1	50
jeditV1	41.0472	0.0212	1	2	2	1	50
jeditV2	86.5800	0.0110	1	2	2	1	50
log4j	1.6563	0.9938	1	2	2	1	50
lucene	17.2450	0.7516	1	2	2	1	50
poiV0	14.7121	0.1871	1	2	2	1	50
poiV1	59.6580	0.2098	1	2	2	1	50
synapse	45.5679	0.6079	1	2	2	1	50
velocity	26.4652	0.9413	1	2	2	1	50
xercesV0	15.4164	0.3131	1	2	2	1	50
xercesV1	30.6858	0.4854	1	2	2	1	50

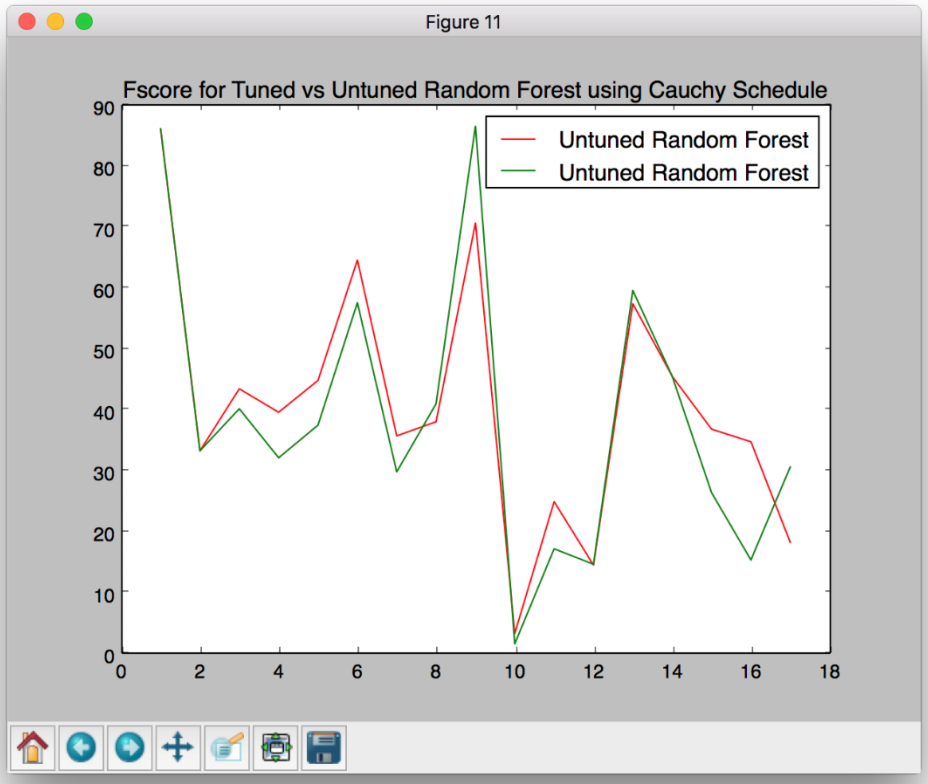

Figure 4.16: Precision for Tuned vs Untuned Random Forest using Cauchy Schedule

G) CART USING PRECISION AS OPTIMISING GOAL WITH SIMULATED ANNEALING

G.1) RANDOM NEIGHBOUR SELECTION

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	82.12891	1	2	8	14	13
antV1	25	1	1	18	17	32
antV2	31.38699	1	1	4	2	13
camelV0	23.97622	1	1	16	18	24
camelV1	28.86204	1	3	15	2	27
ivy	49	1	3	19	6	39
jeditV0	21.93559	1	4	11	14	1
jeditV1	31.64063	1	4	18	7	40
jeditV2	82.64463	1	1	4	9	16
log4j	0.907029	1	3	6	1	35
lucene	11.5533	1	1	7	12	40
poiV0	9.640023	1	2	19	11	47
poiV1	51.16577	1	3	9	18	42
synapse	36.5603	1	1	19	15	5
velocity	19.00066	1	3	11	4	43
xercesV0	10.16593	1	2	7	6	17
xercesV1	8.71398	1	4	17	16	34

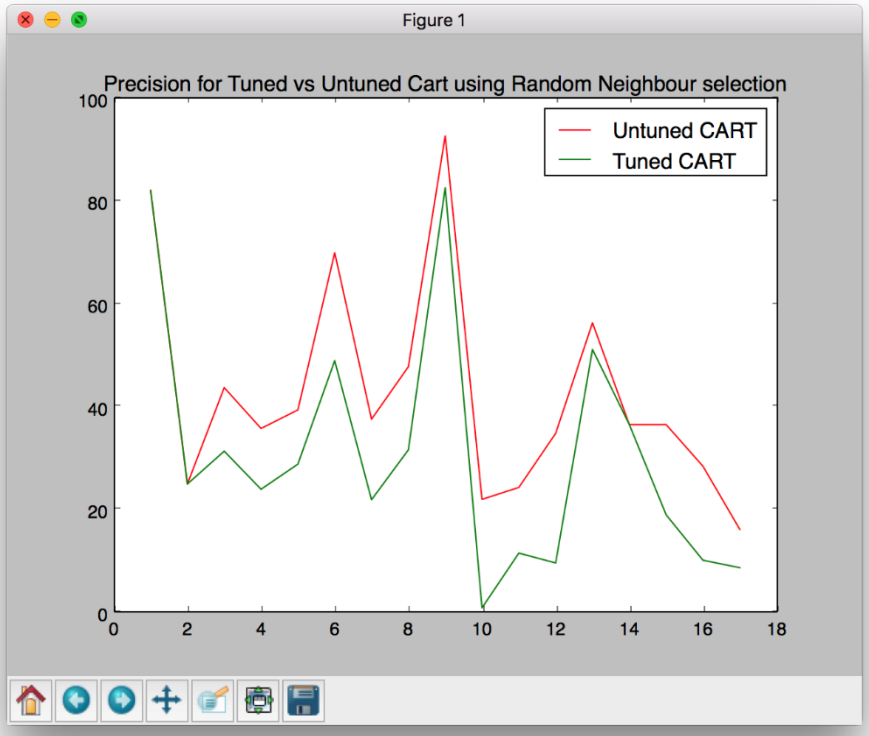

Figure 4.17: Precision for Tuned vs Untuned CART using Neighbour Selection

G.2) FAST SCHEDULING

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	80.16827	0.000247989	1	9	1	9
antV1	25	1	3	9	9	1
antV2	38.40513	0.003123417	2	8	1	50
camelV0	23.97622	0	1	12	7	9
camelV1	28.86204	1	1	2	4	28
ivy	49	1	1	2	4	41
jeditV0	27.43741	0.032832057	1	15	17	10
jeditV1	31.64063	0.890186312	1	2	20	47
jeditV2	90.90909	0.517474232	1	2	1	8
log4j	0.907029	0.567842277	1	2	1	5
lucene	11.5533	0.720077994	4	3	6	1
poiV0	9.640023	0.810396536	1	14	20	13
poiV1	51.16577	0.821256584	1	2	1	1
synapse	36.5603	0.991832136	1	14	9	1
velocity	31.3778	0.125724331	1	2	1	14
xercesV0	31.09775	0.447563802	2	19	1	3
xercesV1	16.20942	0.360236729	1	8	8	16

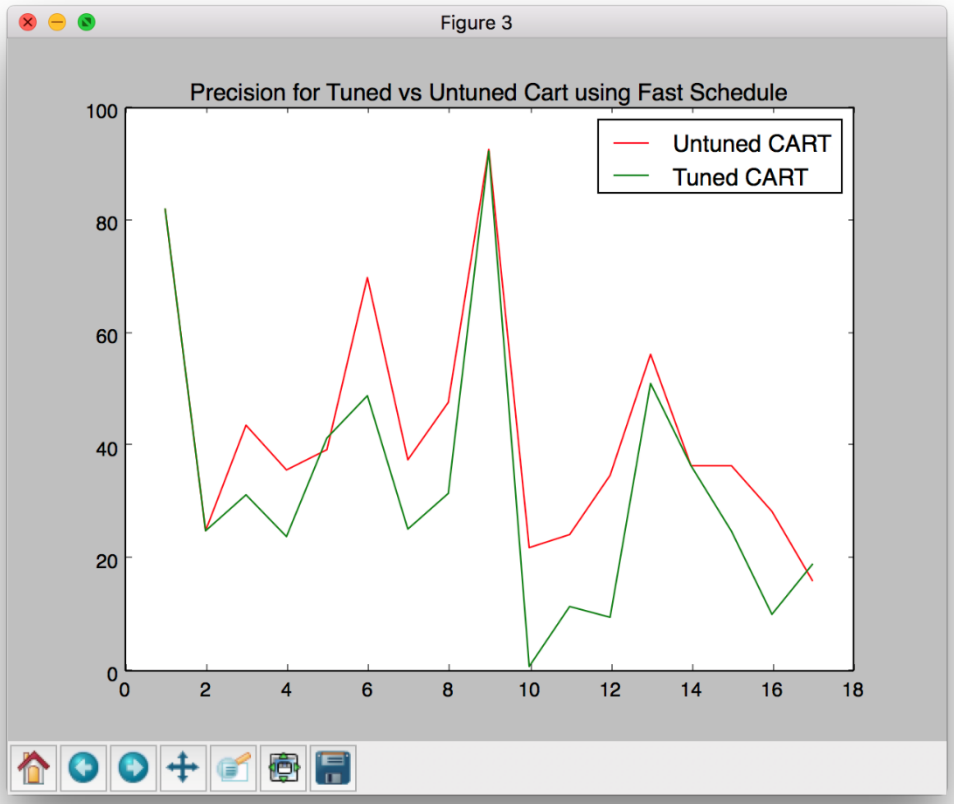

Figure 4.18: Precision for Tuned vs Untuned CART using Fast Schedule

G.3) CAUCHY SCHEDULIING

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	82.12891	0.739481721	1	2	1	1
antV1	25	0.244773274	1	2	1	1
antV2	31.38699	0.614808241	1	2	1	1
camelV0	23.97622	0.369782542	1	2	1	1
camelV1	28.86204	0.618740356	1	2	1	1
ivy	49	0.154771429	1	2	1	1
jeditV0	21.93559	0.984478655	1	2	1	1
jeditV1	31.64063	0.924662596	1	2	1	1
jeditV2	82.64463	0.31372342	1	2	1	1
log4j	0.907029	0.76652787	1	2	1	1
lucene	11.5533	0.937500945	1	2	1	1
poiV0	9.640023	0.950529104	1	2	1	1
poiV1	51.16577	0.053216951	1	2	1	1
synapse	36.5603	0.592423736	1	2	1	1
velocity	19.00066	0.999835572	1	2	1	1
xercesV0	10.16593	0.677628531	1	2	1	1
xercesV1	8.71398	0.817873338	1	2	1	1

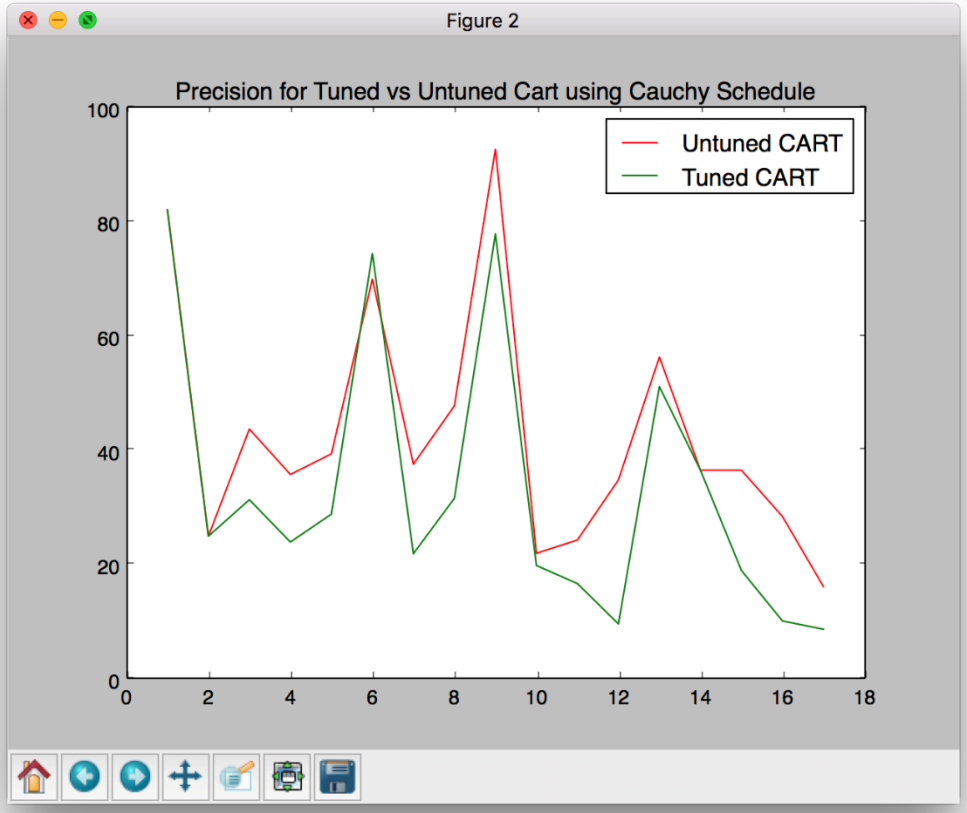

Figure 4.19: Precision for Tuned vs Untuned CART using Cauchy Schedule

H) CART USING F-SCORE AS OPTIMISING GOAL WITH SIMULATED ANNEALING

H.1) NEIGHBOUR SELECTION

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	86.16803	1	1	3	11	23
antV1	33.33333	1	1	15	7	9
antV2	40.23352	1	4	9	15	26
camelV0	32.19029	1	1	11	12	27
camelV1	37.55061	1	4	16	18	35
ivy	57.64706	1	3	19	4	29
jeditV0	29.87778	1	4	6	8	30
jeditV1	40.5	1	3	17	1	43
jeditV2	86.58009	1	4	2	4	8
log4j	1.656315	1	2	7	13	27
lucene	17.245	1	3	5	4	6
poiV0	14.71216	1	2	19	5	9
poiV1	59.65801	1	2	9	8	6
synapse	45.56791	1	4	3	14	5
velocity	26.4652	1	2	8	9	19
xercesV0	15.41647	1	4	15	7	22
xercesV1	13.45586	1	3	16	18	37

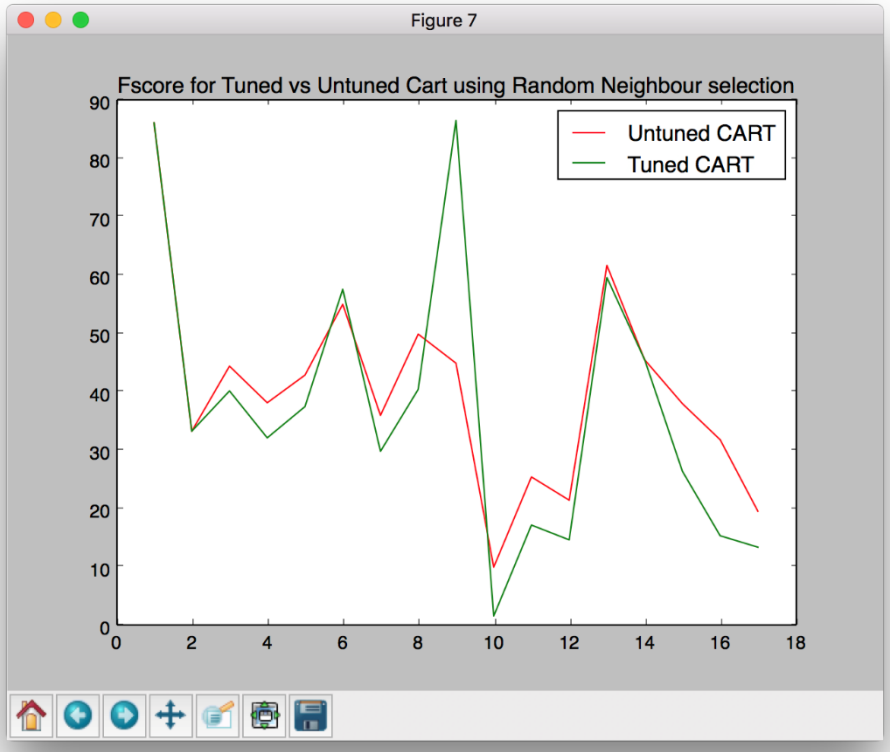

Figure 4.20: F-Score for Tuned vs Untuned CART using Neighbour Selection

H.2) FAST SCHEDULING

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	86.16803	0.768203791	1	2	1	39
antV1	33.33333	1	4	4	6	1
antV2	40.23352	0.197961094	1	15	1	13
camelV0	32.19029	0	1	2	9	12
camelV1	37.55061	0.998795469	1	12	15	1
ivy	50	0.346683318	1	9	6	9
jeditV0	29.87778	1	1	2	4	50
jeditV1	40.5	0.285685924	1	2	1	2
jeditV2	64.17112	0.672463447	1	15	2	1
log4j	11.38402	0	2	14	1	14
lucene	17.245	0.821624142	1	2	17	7
poiV0	14.71216	0.989181024	1	5	3	24
poiV1	59.65801	1	1	2	1	2
synapse	45.56791	0.986657167	1	9	20	49
velocity	41.85868	0.41610147	1	2	1	42
xercesV0	29.40293	0.068849869	1	16	4	1
xercesV1	22.54361	0.020992737	2	2	9	1

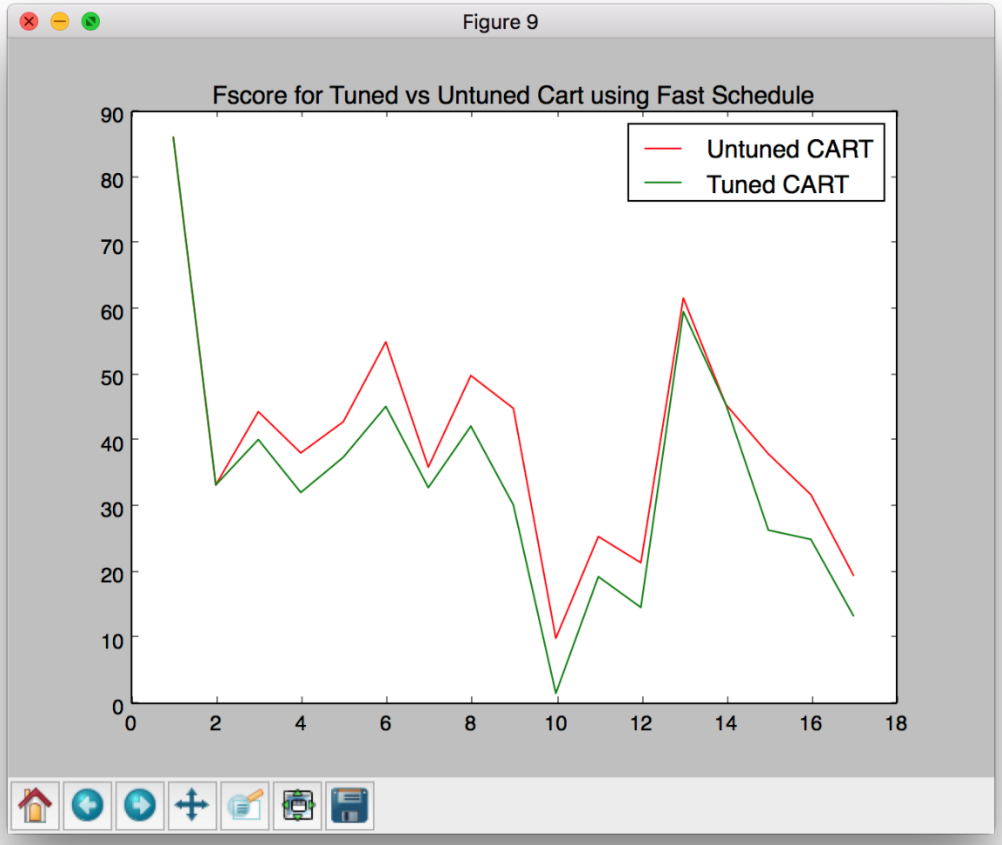

Figure 4.21: F-Score for Tuned vs Untuned CART using Fast Scheduling

H.3) CAUCHY SCHEDULIING

Datasets	Precision	Parameters				
		Threshold	max_feature	max_leaf_nodes	min_samples_split	min_samples_leaf
antV0	81.25	0.202078716	1	2	1	1
antV1	33.33333	0.419896713	1	2	1	1
antV2	40.23352	0.933026729	1	2	1	1
camelV0	32.19029	0.970643737	1	2	1	1
camelV1	37.55061	0.540319942	1	2	1	1
ivy	57.64706	0.995030903	1	2	1	1
jeditV0	29.87778	0.929415087	1	2	1	1
jeditV1	39.80769	0.051382483	1	2	1	1
jeditV2	74.86631	0.014410126	1	2	1	1
log4j	1.656315	0.598233088	1	2	1	1
lucene	18.95931	0.016378722	1	2	1	1
poiV0	14.71216	0.409788511	1	2	1	1
poiV1	59.65801	0.795653421	1	2	1	1
synapse	45.56791	0.583832204	1	2	1	1
velocity	26.4652	0.170391047	1	2	1	1
xercesV0	15.41647	0.718069115	1	2	1	1
xercesV1	19.44126	0.089344041	1	2	1	1

Figure 4.22: F-Score for Tuned vs Untuned CART using Cauchy Schedule

I) SIGNED RANK TEST

Model	Result
Differential Evolution	Null hypothesis rejected (Values are different)
Simulated Annealing	Null hypothesis rejected (Values are different)

Chapter 5 – Conclusion

I) Contributions

In this project, we studied the impact of Parameter Tuning in the field of Software Defect Prediction. Differential Evolution and Simulated Annealing were applied on machine learning algorithms like Random Forest and Classification & Regression Trees (CART) with the aim of optimising Precision or F-Measure. It was found that tuning parameters improved both these performance measures substantially, majorly in the case of differential evolution 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 Simulated Annealing for both Random Forest and 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 Differential Evolution or Simulated Annealing are the best optimizers 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 Learning

[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 .

[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.

[11] K.-L. Du and M.N.S. Swamy, Search and Optimization by Metaheuristics

[12]http://sphweb.bumc.bu.edu/otlt/mphmodules/bs/bs704_nonparametric/BS704_Nonparametric6.html