LEARNING A BRANCHING HEURISTIC FOR SAT SOLVING

CHENXIAO XUE

Department of Mathematics & Computer Science
Davidson College

April 2016

ABSTRACT

The satisfiability problem (SAT) is one of the most intriguing questions in computer science. Although the worst case running time is still believed to be exponential, numerous algorithms have been designed to tackle hard SAT instances efficiently. Many SAT solvers build upon the Davis-Putnam-Logemann-Loveland (DPLL) algorithm, but they all have different ways to pick branching variables.This thesis is primarily concerned with the possibility of using machine learning to determine the best (root) branching variable. We approached this problem by first constructing a regression model to predict the running time of all branching variables. Then we ranked the running time from low to high and chose the branching variable that ranked first to install at the root node. Model performance statistics such as MSE and R^2 suggest that the problem is learnable, despite the fact that our empirical results do not beat MiniSAT, the solver we are trying to improve in this paper.

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [14]

ACKNOWLEDGEMENTS

First and foremost, I would like to express my gratitude to Dr. Ramanujan, without whose guides and expertise this thesis would become entirely impossible. I would also like to thank Dr. Peck for being the second reader of this thesis and providing invaluable opinions.

I am reminded everyday of how lucky I am to have such supportive and wonderful professors around me. Dr. Yerger is my professor, advisor, research partner and friend. I am forever grateful for him not only because he taught me how to play backgammon but also because he opened the research gate for me. As we all know, Dr. Mossinghoff is notorious for assigning hard problems, but it was these hard problems that gave me the most sense of accomplishment. I am thankful to Dr. Heyer for introducing me to project PRONTO. Helping others with my knowledge in math has always been a great pleasure to me. I wish to thank Dr. Ramanujan and Dr. Peck again for introducing me the beauty of computer science. I still remember the nights that I spent trying to draw a picture with turtle or designing an evil hangman.

Ever since I fell in love with math, I tried to take as many courses as I could. However, four years is a short time. I still want to take Numerical Analysis with Dr. Neidinger, Topology with Dr. Molinek, and Advanced Linear Algebra with Dr. Davis. But most important of all, I am thankful to every math professor at Davidson College for showing me how awesome it is to do math.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The boolean satisfiability problem (SAT) is one of the most fundamental problems in computer science. One of the reasons why SAT receives so much attention is that it was the first problem proven to be NP-complete[5]. Because of its conceptual simplicity, many efforts have been dedicated to finding an effective algorithm for solving SAT. Despite its worst-case exponential running time, many SAT solvers are able to solve huge and complex formulas in a fairly short amount of time. This approach has real world applications because problems such as planning, graph coloring, and scheduling can be encoded in SAT. In this thesis, we focus on improving the performance of a particular SAT solver called *MiniSAT*. More specifically, we investigate how machine learning tools could be used to inform heuristics used by MiniSAT.

1.1 BOOLEAN SATISFIABILITY PROBLEMS

1.1.1 *Definitions*

A *propositional logic formula*, also called a *boolean expression*, is built from variables and operators. Legal operators include `AND` (conjunction, denoted by $\wedge$), `OR` (disjunction, denoted by $\vee$), and `NOT` (negation, denoted by $\neg$). Each variable can be assigned only one logical value (i. e. `TRUE` or `FALSE`). A formula is said to be *satisfiable* if it can be made `TRUE` by assigning appropriate values to its variables. The *boolean satisfiability problem* (SAT) is to decide whether a given formula is satisfiable.

A *literal* is either a variable or its negation. If a formula contains a literal without containing its negation, then the literal is said to be a *pure literal*. A *clause* is a disjunction of literals. If a clause contains only one literal, it is called a *unit clause*. A formula is in *conjunctive normal form* (CNF) if it is written as a conjunction of clauses. We focus on formulas in CNF instead of arbitrary formulas because the former can be more easily parsed. A well-established procedure for converting an arbitrary propositional logic formula into a 3-CNF formula (every clause has exactly 3 literals) can be found in [21].

1.1.2 *Examples*

1. An arbitrary formula:

$$x_1 \wedge (x_2 \vee (\neg x_1 \wedge x_2) \vee (x_1 \vee \neg x_2)) \tag{1}$$

2. A CNF formula:

$$(x_1 \vee x_2) \wedge (\neg x_2) \wedge (\neg x_1 \vee x_2) \tag{2}$$

3. A 3-CNF formula:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee x_3) \tag{3}$$

Formula (1) is satisfiable by setting x_1 to `TRUE`, and formula (3) is satisfiable by setting x_3 to `TRUE`. Formula (2) is unsatisfiable because no matter how we assign values to x_1 and x_2, it will always evaluate to `FALSE` in the end. Also, in Formula (2), $(\neg x_2)$ is a unit clause and in Formula (3), literals x_1 and x_3 are pure because $\neg x_1$ and $\neg x_3$ are absent from the formula.

1.1.3 *Algorithms for SAT*

The brute force method for solving a SAT instance is to try every possible combination of assignments. Since each variable can be assigned either `TRUE` or `FALSE`, the running time of this algorithm is $\mathcal{O}(2^n)$. In fact, no polynomial algorithm is known for deciding a SAT problem because that is equivalent to proving $P = NP$[1][21].

There is an annual SAT competition intended to identify new challenging benchmarks and to promote new solvers for the boolean satisfiability problem as well as to compare them with state-of-the-art solvers[19]. These state-of-the-art solvers, although differing in details, primarily apply two types of algorithms: a variant of the **Davis-Putnam-Logemann-Loveland(DPLL)** algorithm and a **stochastic local search** algorithm. The former is a complete algorithm while the latter is an incomplete one. A complete algorithm always yields satisfiable or unsatisfiable when it is finished. An incomplete algorithm will eventually find a satisfying assignment if one exists, but it cannot prove that a formula is unsatisfiable. Despite the worst-case exponential runtime, these solvers can often efficiently solve hard structured problems with over a million variables and several million constraints [12].

DPLL

The DPLL algorithm is a complete, backtracking-based algorithm. The algorithm details are presented in Figure 1[12].

A literal in a unit clause has to be set to `TRUE` in order for the entire formula to be `TRUE`. This is accomplished by the function `Unit-Propagate`, which eliminates all the unit clauses . After the literal is set to `TRUE`, the function removes any clause that contains this literal (because the clause is satisfied). It also removes

1 There are, however, special classes of formulas that can be solved polynomially, for example, 2-SAT and Horn-SAT[7].

```
function DPLL($\phi$):
    if $\phi$ contains no clauses then
        return TRUE
    end if
    if $\phi$ contains an empty clause then
        return FALSE
    end if
    $\phi \leftarrow$Unit-Propagate($\phi$)
    $\phi \leftarrow$Pure-Literal-Assignment($\phi$)
    $x \leftarrow$ a variable in $\phi$
    return DPLL($\phi \wedge x$) or DPLL($\phi \wedge \neg x$)
end function

function Unit-Propagate($\phi$)
    while $\phi$ contains a unit clause do
        $l_u \leftarrow$ the literal in a unit clause
        for every clause $c$ in $\phi$ do
            if $c$ contains $l_u$ then
                Remove $c$ from $\phi$
            else if $c$ contains $\neg l_u$ then
                Remove $\neg l_u$ from $c$
            end if
        end for
    end while
    return $\phi$
end function

function Pure-Literal-Assignment($\phi$)
    while $\phi$ contains a pure literal $l$ do
        Remove any clause that contains $l$
    end while
end function
```

Figure 1: Functions in the DPLL algorithm

the negation of this literal from all clauses because they do not contribute to a satisfying assignment. Note that the while loop in the function could cause a cascade of unit propagations. Take Formula (2) as an example. The first iteration of unit propagation sets x_2 to `FALSE`. As a result, both $(x_1 \vee x_2)$ and $(\neg x_1 \vee x_2)$ become unit clauses. Based on $(x_1 \vee x_2)$, we should assign `TRUE` to x_1, and $(\neg x_1 \vee x_2)$ becomes an empty clause. At this point, we can conclude that Formula (2) is unsatisfiable.

The function `Pure-Literal-Assignment` removes any clause that contains a pure literal because all pure literals can be assigned `TRUE` without incurring contradictions.

A complete, backtracking-based algorithm built upon the DPLL algorithm is the **Conflict-Driven Clause Learning (CDCL)** algorithm [2]. In the past two decades, the performance of SAT solvers has improved dramatically due to the advent of CDCL. The biggest difference between the DPLL algorithm and the CDCL algorithm is that the latter uses non-chronological backtracking [2], which significantly reduces the search space. The algorithm learns and remembers conflicts by adding new clauses. This increases the formula size, but the learned clauses may quickly find a contradiction if we have an unsatisfying assignment. Another technique utilized by both DPLL and CDCL is random restarts. When restarting, the solver discards all previously made decisions but keeps the learned clauses. The common belief is that restarts help the solver avoid spending excessive time in branches with no easy-to-find solution [18]. In this thesis, we focus on a solver called *MiniSAT* that implements the CDCL algorithm [8].

Stochastic local search

The stochastic local search algorithm is an incomplete algorithm that starts with a random assignment of all variables. Consider the following formula:

$$(x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_2 \vee \neg x_3).$$

Suppose all variables are assigned `TRUE` in the beginning. This assignment is not a satisfying assignment because the clause $(\neg x_2 \vee \neg x_3)$ is unsatisfied. Now we randomly flip the assignment of a variable in an unsatisfied clause, say x_2. Now, $(\neg x_2 \vee \neg x_3)$ is satisfied but $(\neg x_1 \vee x_2 \vee \neg x_3)$ becomes unsatisfied. Suppose we now flip x_1. Then we get a satisfying assignment for the entire formula. Note that the choices should not be entirely

2 In non-chronological backtracking, the program does not backtrack to the previous level when detecting an unsatisfying assignment. Instead, it backtracks to the last branch that causes the conflict.

random because we do not want to flip x_2 back to `TRUE` in the second iteration. In practice, most stochastic local search algorithms follow heuristics to minimize runtime. Stochastic local search is often faster than DPLL, but it is incomplete.

1.2 MACHINE LEARNING

1.2.1 *Overview*

In 1959, Arthur Samuel defined machine learning as a "field of study that gives computers the ability to learn without being explicitly programmed" [20]. In this work, we primarily use techniques from an area of machine learning known as supervised learning.

In supervised learning, the computer is presented with inputs (features) and the corresponding outputs (targets). The goal is to learn the relation between features and targets so that it can predict the outputs for new input data.

Supervised learning problems can be further divided into two types of problems: **regression** and **classification** problems. In a regression problem, the output variable takes on continuous values, whereas in a classification problem, the output variable takes on discrete values. A variable x is said to be *continuous* if the following condition holds: if x can take on values x_1 and x_2, then for any $x_1 < x_3 < x_2$, x can also take on the value x_3. Examples of regression problems include predicting a student's GPA given how much he studies. Examples of classification problems include identifying handwritten English letters. Classification problems are sometimes approached by using regression (logistic regression). Regression problems can also be converted into classification problems by introducing artificial classes [1]. Numerous algorithms for supervised learning exist (linear regression, logistic regression, K-nearest neighbor search, etc.) and we use the ones implemented by a Python library called `scikit-learn` [17].

2 RELATED WORK

Although SAT has received a lot of attention from the research community, to the best of our knowledge few papers relate it to machine learning. In this section, we focus on two such papers and discuss their methodology as well as the results.

2.1 SATZILLA

SATzilla is a solver that competed in the SAT competition in 2007 and won several medals [23]. As opposed to other modern solvers, SATzilla is not a single solver, but rather a collection of solvers. This collection consists of SAT solvers, such as MiniSAT, that performed well in the past SAT competitions. Given an input instance, SATzilla first runs all solvers in parallel briefly to make sure that the given instance cannot be solved within a fixed amount of time. Then, it computes formula features, such as the fraction of binary and ternary clauses[1], and feeds these features into a pre-trained model to determine which SAT solver to use for a long running job[2]. The reason for building an ensemble solver comes from the observation that a single SAT solver can be incredibly efficient at one instance type but inefficient at another.

In [23], the authors divide the training instances into two categories, namely, satisfiable instances and unsatisfiable instances. However, when testing performance on the test set, they cannot know beforehand whether an instance is going to be satisfiable or unsatisfiable. Therefore, they use a model that predicts the likelihood of a formula being satisfiable using the same features. The classification model performs well on random instances and industrial instances[3] with about 90% accuracy.

SATzilla won the first place in both Random and Handmade categories. It also placed second in the Industry category.

2.2 PERCEPTRON LEARNING

Flint and Blaschko have used the perceptron learning algorithm to study formulas that are solvable in polynomial time [10]. Given a (polynomial time solvable) formula, the perceptron decides whether

1 A binary clause has two literals and a ternary clause has three literals
2 All features have polynomial computing time, so this process does not take a long time
3 Industrial instances are obtained from encoding industrial problems into SAT.

a certain node (an assignment of a variable) yields a shortest proof of satisfiability. To generate training data, the authors searched the entire formula space. That is, they computed whether a node was 'good' for every single node of the formula. The features used are similar but not the same as the ones used in SATzilla. In particular, all features used in the Perceptron Learning paper are polynomially computable.

While SATzilla is based on a practical solver, Flint and Blaschoko use machine learning in a more theoretical way. To some extent, the approach of [10] is similar to that of this thesis because it tries to predict some attributes of branching variables. What separates it is that Flint and Blaschoko focused on predicting whether a given assignment of a branching variable leads to a satisfiable formula. Although it does not calculate the runtime directly, it is by no means irrelevant, because if we are able to choose the correct variable every time, we can reduce the runtime wasted on backtracking. In fact, they replace the MiniSAT solver's branching heuristic with their perceptron and the running time on graph coloring and hardware verification is improved by an order of magnitude.

Since each literal is considered individually, the features have to be computed at each literal and at each step. Therefore, the total runtime for computing the labels of a training set is exponential. This limits the number of data points that can be used. Despite limitations on the size of the training set, the performance on the test sets is good. For planar graph coloring problems, this algorithm converged after only 40 iterations (for some fixed parameters of the graph) [10].

3 EXPERIMENTS

3.1 OVERVIEW

Earlier we mentioned a solver that performs well on a certain family of instances may perform very poorly on another. Although Xu et al.[23] built an ensemble solver that chooses different solvers for different types of instances, the underlying mechanism for each solver is still hard-coded. More specifically, there is a fixed way of choosing the next branching variable. In this paper, we investigate the possibility of handing this job, i.e., choosing branching variables, to a computer. To accomplish this goal, we need to build an 'oracle' that can predict the (average) runtime after we choose to branch on some variable. Ideally, the features that determine this runtime should be inexpensive to compute.

3.2 DATA COLLECTION

Because MiniSAT is already a highly optimized solver, we will not be able to improve performances on simple formulas. To make a formula hard, we need to have enough variables and control the clause-to-variable ratio. It is shown int [16] that for random 3-CNF formulas, a clause-to-variable ratio of approximately 4.2 gives the hardest instances. In this work, we focus on random 3-CNF formulas with 300 variables and 1,250 clauses. In a random 3-CNF formula, each clause is constructed by picking 3 elements uniformly at random from the variable set without replacement and negating them with a probability of 0.5. We obtained a total of 101 formulas from the Tough SAT Project website[22]. We use two kinds of features in constructing our dataset: variable-specific and formula-specific features.

3.2.1 *Variable Features*

Since our goal is to predict which variable to pick, the features of our dataset have to be variable-specific. Hence, we decide to first build a **variable incidence graph** (VIG) and use centrality measures as our features. In a VIG, every variable has its corresponding vertex, and two vertices are adjacent if they appear in the same clause. Normally, the edges are weighted but we use a simpler version where each edge has a weight of 1. After the graph is constructed, we use the built-in functions of the `networkx` Python library to compute the following five features for all vertices [13]: [1]

1. Degree Centrality: the *degree centrality* for a node v is the fraction of nodes in the graph it is connected to.

2. Closeness Centrality: the *closeness centrality* of a node u, denoted $C_C(u)$, is the reciprocal of the sum of the shortest path distances from u to all $n-1$ other nodes [11]. Since the sum of distances depends on the number of nodes in the graph, closeness is normalized by the sum of the minimum possible distances n-1. That is,

   $$C_c(u) = \frac{n-1}{\sum_{v=1}^{n-1} d(v,u)},$$

   where $d(v,u)$ is the shortest-path distance between v and u, and n is the number of nodes in the graph.

3. Betweenness Centrality: the *betweenness centrality* of a node v, denoted $C_B(v)$ is the sum of the fraction of all-pairs shortest paths that pass through v

   $$C_B(v) = \sum_{s,t \in V} \frac{\sigma(s,t|v)}{\sigma(s,t)},$$

   where V is the set of nodes, $\sigma(s,t)$ is the number of shortest (s,t)-paths, and $\sigma(s,t|v)$ is the number of those passing through some node v other than s,t. If $s=t$, then $\sigma(s,t)=1$, and if $v \in \{s,t\}$, then $\sigma(s,t|v)=0$ [4].

4. Eigenvector Centrality: the *eigenvector centrality* computes the centrality for a node based on the centrality of its neighbors. The eigenvector centrality for node i is given by the i^{th} entry of the eigenvector corresponding to the largest eigenvalue of the adjacency matrix [3]. Let A be the adjacency matrix of the graph G with eigenvalue λ,

   $$\mathbf{Ax} = \lambda \mathbf{x},$$

   then the eigenvector centrality for node i is given by the i^{th} component of x. By virtue of the Perron-Frobenius theorem[**chang2008perron**], there is a unique and positive solution if λ is the largest eigenvalue associated with the eigenvector of the adjacency matrix A.

5. Communicability Centrality: the *communicability centrality*, also called subgraph centrality, of a node n is the sum of closed walks of all lengths starting and ending at node n [9].

1 These features are polynomially computable.

3.2.2 *Formula Features*

In addition to variable features, we also use formula-wide features. To transform formula features into variable features, we simply take the formula features after we set a variable. For example, if we set x_1 to TRUE, then all clauses that contain the literal x_1 will be deleted and all appearances of $\neg x_1$ will be removed from the formula. We do not perform any pure-literal assignment or unit propagation when calculating the formula features after setting a variable. Since there are two values that can be assigned to a variable, we will obtain two sets of features for each variable, one obtained by setting this variable TRUE and the other obtained by setting it FALSE. The formula features of a variable are simply the concatenation of these two sets.

The formula features used are a subset of the features used in [23]:

1. Number of Binary Clauses
2. The max, min, mean, and standard deviation of variable degree in the VIG.
3. The max, min, mean, and standard deviation of variable degree in the VCG. [2]
4. The max, min, mean, and standard deviation of fraction of positive literals of a variable.
5. The max, min, mean, and standard deviation of fraction of positive literals of a clause.

3.2.3 *Target Variable*

The target variable in our study is the expected runtime of MiniSAT after we install a specific variable at the root of the search tree. Technically, the empirical runtime also depends on how we assign the chosen variable, so we hedge our bets by predicting an "expected runtime", given a random assignment of TRUE/FALSE. Suppose we pick a variable and randomly assign a value to it. If the original formula is satisfiable under the current assignment, then we do not need to check the other branch. However, if it is unsatisfiable, we need to assign the opposite value to the selected variable and run the solver again. Suppose that the runtimes of the two branches are t_1 and t_2 and that the outcomes(valuations) under t_1 and t_2 are V_1 and V_2, respectively. The method to calculate the expected runtime is summarized in Table 1. If both V_1 and V_2 are TRUE, then no matter which branch we pick, the solver will terminate without trying the other branch. Now suppose only one of them is TRUE, say V_1. If the solver

2 In a *Variable Clause Graph* (VCG), each vertex represents either a variable or a clause. A variable is connected to a clause if it appears in this clause.

chooses t_1's branch, it will not try the other branch because the formula is satisfied. However, if the solver chooses t_2's branch first, it has to backtrack and search the other branch, too. Hence, the expected runtime is $t_1 + t_2/2$. Finally, if both V_1 and V_2 are FALSE, the entire formula is unsatisfiable, and both branches have to be checked.

V_1	V_2	formula
TRUE	TRUE	$\frac{t_1}{2} + \frac{t_2}{2}$
TRUE	FALSE	$t_1 + \frac{t_2}{2}$
FALSE	TRUE	$\frac{t_1}{2} + t_2$
FALSE	FALSE	$t_1 + t_2$

Table 1: The expected runtime under different circumstance.

Figure 2 displays the distribution of the calculated expected runtime (in seconds) of a single randomly picked formula. Since runtime has a lower bound of 0, the distribution is skewed. Hence, we take the log of runtime to obtain a normal distribution and use this in our dataset.

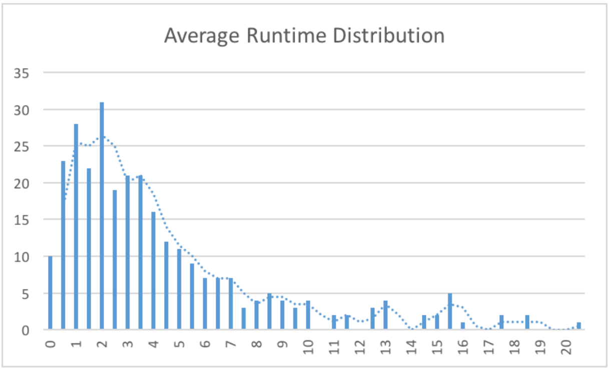

Figure 2: Expected runtime distribution.

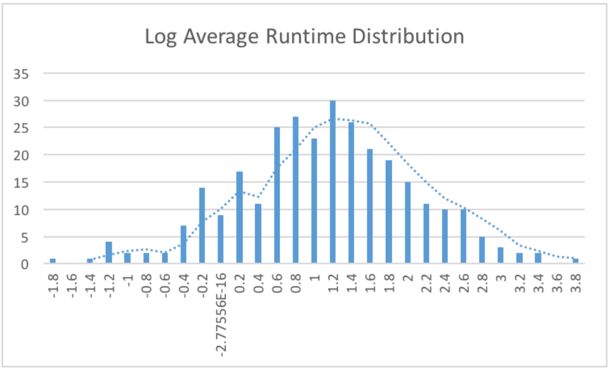

Figure 3: Log of expected runtime distribution

Mean-normalization is a standard data preprocessing step to scale data so that all features have a mean of 0 and a standard deviation of 1. For each feature x, we compute the mean-normalized value x':

$$x'_i = \frac{x_i - \bar{x}}{\sigma(x)},$$

where $\bar{x}$ is the mean of this feature and $\sigma(x)$ is the standard deviation of this feature. All variables are mean-normalized within formula. That is, the mean $\bar{x}$ and the standard deviation $\sigma(x)$ used in mean-normalization are derived from only the 300 data entries corresponding to a single formula.

3.3 MODEL SELECTION

We investigated two different algorithms for building our regression models: **Ridge Regression** and **Random Forest Regression**.

3.3.1 *Ridge Regression*

One of the simplest regression models is linear regression. In a linear regression model, the output is a linear combination of the input features:

$$\hat{y} = \sum_{i=1}^{n} \theta_i x_i,$$

where θ_i's are parameters to learn, x_i's are input features, and n is the number of features. To find the best fit line, i.e. the best set of parameters $\vec{\theta}$, we minimize the *cost function* $J(\vec{\theta})$

$$J(\vec{\theta}) = \sum_{i=1}^{m} (y^{(i)} - \hat{y}^{(i)})^2,$$

where $y^{(i)}$ is the true label of the i^{th} data entry, and $\hat{y}^{(i)}$ is the prediction given by the model. There are many ways to find the minimum of $J(\vec{\theta})$. For example, one could use gradient descent or solve for an analytical solution.

Ridge regression is an extension of linear regression, that incorporates a scheme called **regularization**. The idea behind regularization is that simpler theories (models) are preferable to more complex ones [24]. One reason for such preference is that complex models tend to overfit.[3] Hence, ridge regression penalizes the sizes of all parameters by adding a regularization term to the cost function:

$$J(\vec{\theta}) = \sum_{i}^{m} (y^{(i)} - \hat{y}^{(i)})^2 + \lambda \sum_{i}^{n} \theta_i^2,$$

3 Overfitting means that the model captures random error or noise in the training set and fails to generalize effectively to new data.

where λ is the *regularization factor*. This is also referred to as the L_2 regularization owing to the use of L_2-norm of $\vec{\theta}$. Consequently, if we compute $\arg_{\vec{\theta}} \min J(\vec{\theta})$, all parameters will be shrunk.

3.3.2 *Decision Trees*

Decision trees are commonly used for classification. One interesting game that uses the same technique as decision trees asks you a series of questions to guess a celebrity you are thinking of; however, it is also possible to use decision trees to solve regression problems [6].

To use decision trees to make predictions, we first need to grow a tree. The basic idea of growing a tree is to find the split that results in maximal purity of the children nodes. A node is said to be *pure* if it contains only one kind of label. For example, suppose we have the dataset shown in Table 2 and we want to predict whether a person wears glasses. There are two splits we can make because there are two distinct features. The first split is by gender. However, after splitting the dataset, each smaller set still contains both 'Y' and 'N'. Therefore, the children nodes are not pure. The second split is by whether employed. If we use this split, both children nodes are 100% pure because they only contain one label or the other.

GENDER	EMPLOYED	WEAR GLASSES
Female	Y	Y
Male	N	N
Female	N	N
Male	Y	Y

Table 2: Fictitious training data for learning a decision tree.

Typically, the first split is not going to guarantee 100% purity, so we need to keep growing the tree. Essentially, you treat each child node as the root node in an input space where one feature has been eliminated. The growing process terminates when the nodes become entirely pure or when we run out of features to split the data. Sometimes, to speed up the process and to avoid overfitting, we allow a certain degree of impurity or set a threshold for the depth of the tree. The degree of impurity that can be tolerated is dependent on the planned running time, the precision of prediction, etc. There are also pruning methods to reduce the complexity of the final model [15].

When new data comes in, we start from the root node and traverse the tree according to features of the example. When we arrive at a leaf, the prediction is given by the majority of the labels in the leaf node.

3.3.3 *Random Forest Regression*

Random forest regression is an **ensemble method**. An ensemble method uses the combination of several models to make predictions. The benefit of such approach is to decrease the variance produced by a single model, while maintaining the same expected output (thus introducing no extra bias). A random forest regressor aggregates several decision trees. More specifically, it first builds a given number of decision trees whose splitting criterion at each node is chosen randomly. Then it solicits a prediction from each decision tree. Finally, the model aggregates the results by taking the mode if the problem is a classification one, or the mean if it is a regression one.

In random forest regression, we usually do not restrict the depth or prune trees because aggregating multiple trees already reduces the chance of overfitting. Hence, the only thing that we need to control is the number of trees or estimators to be used. Theoretically, a larger number is preferable because this number has a negative correlation with variance. Empirically, the model will converge after some number and a further increase will only result in a longer runtime. Therefore, to pick the right number of estimators, we slowly increase it until the model converges.

3.4 PROCEDURE

We applied the following training/testing protocol:

1. split the data set into three subsets.
2. perform regression/classification analysis on the first subset (**training set**).[4]
3. tune hyper-parameters, such as the regularization factor in ridge regression, using the second subset (**validation set**).
4. report error on the third subset (**test set**) once the parameters and hyper-parameters are chosen.

Splitting the dataset into three parts may reduce the size of the training set significantly, thus compromising the accuracy of the model. One way to solve this problem is to use a method called **cross-validation**. In cross-validation, the training set and the validation set are combined together. Multiple iterations are required. During each iteration, a fixed portion of this combined set is used as the validation set and the rest as the training set. Different iterations use different portions of this combined set as the validation set. After all iterations, each

4 Training set is usually the largest in size to guarantee a good performance of the model.

data entry is used exactly once as a part of a validation set. The reported error would be the average error of all iterations. Two popular cross-validation methods are **K-fold** and **Leave-One-Out**. In a K-fold cross-validation, the training set is split into K folds of similar sizes. Each iteration takes one fold as the validation set and the remaining $K - 1$ folds as the new training set. In a Leave-One-Out cross-validation, each data entry is considered as a validation set and the number of iterations is equal to the size of the training set.

3.5 FITTING PROBLEM

A model that does not fit well to the training set can be troublesome because it lacks predictive power. This is usually called **underfitting**. However, a model that fits very well to the training set can also be problematic because it **overfits** the data. In other words, a model that overfits absorbs stochastic changes into its underlying model. Figure 4 shows how underfitting and overfitting look like. The left figure demonstrates the problem of underfitting because the gaps between the model predictions and the actual results are huge. The right one has an overfitting problem because the model is too complicated and will fail to generalize. The middle one is neither underfitting nor overfitting.

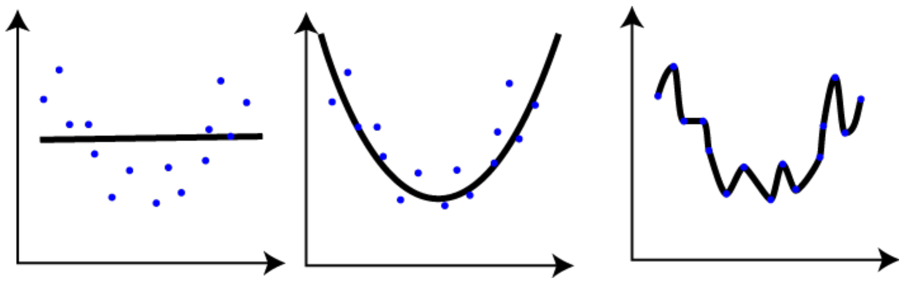

Figure 4: Underfitting and overfitting.

To detect when overfitting occurs, we look at the error of both the training set and the test set as shown in Figure 5.[5] As the number of iteration increases, we expect to see a gradual decrease in the error of the training set. If the model does explain the data better and better with each iteration, the error of the test set is also going to decrease. Hence, overfitting occurs when we see rises in the error of the test set. Therefore, one way to prevent overfitting is to calculate the error on the validation set and stop iterating as soon as the error on the validation set begins to rise.

5 We treat the validation set as the test set.

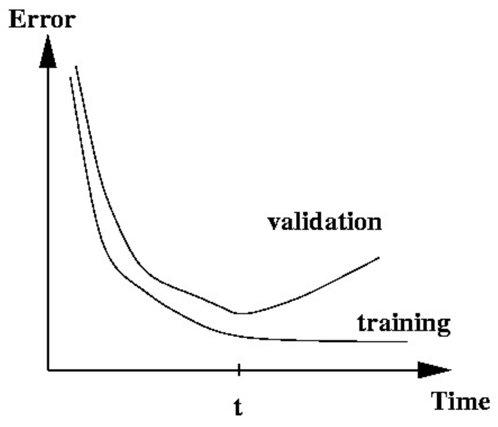

Figure 5: Error of training and test sets vs. the number of iterations.

4 RESULTS

In this section, we test our models by measuring their model performances and solver performances. The model performances are given by *mean squared error* and *R-squared*. The solver performances are measured by empirical running time.

4.1 PERFORMANCE METRICS

Testing model performance is as important as constructing a model because we want to pick the best model of all constructed models. Depending on whether the problem is a classification or a regression problem, we use different statistics to test models. For a classification problem, we can calculate the **accuracy** of predictions, which is the number of times the outcomes are correctly predicted in the test set. Sometimes, however, accuracy is not a good measure. For example, suppose we are predicting whether a person is going to develop leukemia and are we draw samples randomly from the entire population. Then approximately 98% of the data labels are negative, which means that a dummy model that always predicts negative will have an accuracy of 98%. This kind of model is not going to be helpful in any practical sense. An alternative way to measure performance is to calculate the **precision** and **recall** of the model. Precision and recall are defined as:

$$\mathrm{Precision} = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fp}} \qquad \mathrm{Recall} = \frac{\mathrm{tp}}{\mathrm{tp}+\mathrm{fn}},$$

where tp, fp, and fn are defined in the confusion matrix represented by Table 3. Intuitively, precision measures how likely the sample is

	Predicted positive	Predicted negative
Condition positive	True positive(tp)	False negative(fn)
Condition negative	False positive(fp)	True negative(tn)

Table 3: A confusion matrix.

going to be positive when the model predicts positive and recall measures the percentage of positive samples that are accurately predicted. Using either one of the two metrics individually is insufficient because a model that always predicts negative will have a precision of 1 and a model that always predicts positive will have a recall of 1.

Now if the problem is a regression problem, we can use two different statistical measures to calibrate performance. The first is **mean**

squared error, given by $\mathrm{MSE} = \frac{1}{m}\sum_{i=1}^{m} w_i(y_i - \hat{y}_i)^2$, where m is the total number of samples in the test set, w_i is the weight and is usually 1, y_i is the actual value of the i^{th} sample, and $\hat{y}_i$ is the predicted value. The second is **coefficient of determination**, also known as the **R-squared value**, given by $R^2 = 1 - \frac{\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{m}(y_i - \bar{y})^2}$, where $\bar{y}$ is the average of the target values of all samples. Translated in English, R^2 is the variation from prediction over the variation from sample mean (subtracted from 1). Note that a perfect model should have $\mathrm{MSE} = 0$ and $R^2 = 1$ and a baseline model that always predicts $\bar{y}$ should have $R^2 = 1$. Negative R^2 is possible because a model can be arbitrarily worse.

4.2 MODEL PERFORMANCE

We calculated both MSE and R^2 for both the ridge regressor and the random forest regressor. When calculating these, we used the same training set and test set, but the training set and test set themselves were randomly selected from the entire dataset.

Table 4 summarizes the results for both models. According to our results, the random forest regressor is a better model than the ridge regressor. This contradicts to Xu et. al's claim that the ridge regressor is better [23], but it could be because we used a different set of features and trained the models on a different distribution.

MODEL	MSE	R-SQUARED
Ridge	7745.84	0.2826
RandomForest	1888.94	0.8251

Table 4: MSE and R^2 for both models.

First, note that a model that always predicts the average of all targets has a R-squared value of 0 and a model that always correctly predicts the targets has a R-squared value of 1. Both our models are clearly better than the baseline model of always predicting the mean, so they clearly have learned something from the data. Because our models are trained on the training set and tested on the test set, the calculated statistics should generalize well to the entire distribution. Now that we have non-trivial models, we would like to put them to use, i.e., let them decide which branching variables we should pick for the root node.

4.3 SOLVER PERFORMANCE

In this section, we build models by running regression on 101 formulas. Then we test them on 150 new formulas drawn from the same distribution. Each model will return a branching variable for a formula. [1] We force the MiniSAT solver to branch on the indicated variables at the root node but hand the rest of the problem over to MiniSAT. That is, we only control the first step of branching. Finally, we obtain the running time measurement in each case and compare it to the original MiniSAT running time. The original running time is obtained by running MiniSAT without setting the first branching variable. Hence, MiniSAT makes the first move according its default behavior.

The solver performances were measured on both absolute and relative statistics (separated by a horizontal line in the tables). For the absolute statistics, we measured, of all 150 formulas, how many times the branching variable is ranked in the 50^{th} and in the 10^{th} percentile of all variables in the same formula. We also measured the mean and the median of both the rank and the running time. For the relative statistics, we measured how many times a model beats another model on runtime.

Statistics are summarized in Table 5. The runtime statistics do not reveal much other than the fact that MiniSAT's default heuristic has better performance. The rank statistics, however, can be compared to a baseline model that selects branching variables randomly. If we assume that it is a uniform distribution, then the expected values and standard deviations (in parenthesis) can be calculated, as presented in the last column of Table 5. We calculated the first two statistics (ranked in the 50^{th} and in the 10^{th} percentile) assuming that the distribution is binomial. The number of independent experiments n is 150, and the probability of success p is 0.5 for the first statistic and 0.1 for the second. Hence, the expected value can be calculated by np and the standard deviation is $\sqrt{np(1-p)}$. The next two statistics were calculated under the assumption that the distribution is uniform. A uniform distribution with a lower bound a and an upper bound b has expected value $\frac{a+b}{2}$ and standard deviation $\frac{b-a}{2\sqrt{3}}$. Statistics show that in a single instance, the heuristics given by MiniSAT and our models are not significantly better randomly picking a branching variable; however, collective data obtained from all 150 formulas suggest that the improvement is significant.

As for relative statistics, a number greater than 75 means the model represented by this column is better than the model represented by this row. According to Table 5, MiniSAT > Ridge Rigression> Random Forest Regression on these 150 formulas.

1 The model actually returns a list of running times for each variable, and we take the variable that corresponds to the minimum running time to be the branching variable.

CRITERION	MINI	RANDOM	RIDGE	BASELINE
ranked in the 50th	102	85	87	75 (6.12)
ranked in the 10th	39	21	21	15 (3.67)
mean(rank)	105	133	127	150.5 (86.31)
median(rank)	93	123	115	150.5 (86.31)
mean(time)	12.80	15.31	15.34	
median(time)	3.505	3.775	4.245	
beats Mini	x	67	65	
beats Random	83	x	83	
beats Ridge	84	66	x	

Table 5: Solver performance of Mini(MiniSAT), Random(Random Forest Regression), and Ridge(Ridge Regression) on all instances.

We then filtered the formulas so that only hard formulas were considered. A formula is characterized to be "hard" if the running time of MiniSAT exceeds 10 seconds no matter which model we choose. Only 37 out of 150 formulas are considered hard by this criterion. We then calculated the same statistics for the two models on these 37 formulas. The results are summarized in Table 6. Now MiniSAT no longer has the best performance in all criterion but still takes the shortest amount of time overall. The overall significance went up notably for both Random Forest and Ridge but only slightly for MiniSAT. Now in relative statistics, a number greater than 18.5 indicates outperformance. Therefore, there exists an intransitive relationship between the three models, where MiniSAT > Ridge Regression > Random Forest Regression > MiniSAT.

CRITERION	MINI	RANDOM	RIDGE	BASELINE
ranked in the 50th	26	26	29	18.5 (3.04)
ranked in the 10th	10	12	6	3.7 (1.82)
mean(rank)	93	101	88	150.5 (86.31)
median(rank)	77	57	70	150.5 (86.31)
mean(time)	42.63	45.57	43.63	
median(time)	32.53	39.51	36.53	
beats(Mini)	x	20	18	
beats(Random)	17	x	20	
beats(Ridge)	19	17	x	

Table 6: Solver performance of Mini(MiniSAT), Random(Random Forest Regression), and Ridge(Ridge Regression) on hard instances.

4.4 CONCLUSION

As shown in earlier sections in this Chapter, although our model has very high R^2 value, the solver performance is still not good enough to beat MiniSAT's built-in heuristic. What is even more surprising is that while Random Forest Regression has significantly better R^2 and MSE than Ridge Regression, its solver performance is only as good as, if not worse than, Ridge Regression. One explanation is that choosing the best branching variable is different from predicting runtime because a slight deviation from the actual runtime will result in completely different decisions. For example, when we increased the number of estimators for Random Forest Regression from 200 to 400, while R^2 and MSE stayed fairly constant, the chosen branching variable changed for a lot of the formulas. Another explanation is that learning takes place on a completely different set of formulas than the ones for testing solver performance. When calculating model performances, although the training set and the test set were disjoint, the entries were shuffled so the training set and the test set most likely contain samples from the same formulas.

In the solver performance section, when we shift our attention to hard instances, the performance of both models is suddenly boosted while MiniSAT's performance stays almost the same. One conjecture is that these hard instances are mostly formulas that are unsatisfiable because unsatisfiable formulas tend to take a longer time. If this were true, then it would mean that our models learned better on unsatisfiable formulas. Therefore, one way to improve the solver performances as well as the model performances is to build models separately for satisfiable and unsatisfiable formulas.

4.5 FUTURE WORK

Both model performance and solver performance have shown that the problem is clearly learnable. The following things could be done to potentially enhance performance:

1. Use a larger set of features. For example, we could include variable activity computed by MiniSAT for each variable, or the probing features presented in Xu et. al [23].

2. Train on industrial instances instead of random ones. We conjecture that industrial instances have more variance in its features and therefore are easier to learn.

3. Re-run regression after each decision is made (or a certain number of decisions are made).

4. Filter out easy formulas.

5. Train separately on satisfiable and unsatisfiable instances. Note that when a new formula come in, we do not know whether it is satisfiable. Therefore, we need to first train a classification model that determines whether a formula is satisfiable or not. The features could be the same as those used in the regression model.

BIBLIOGRAPHY

[1] Anton Andriyashin. "Financial applications of classification and regression trees." In: (2005).

[2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. "Conflict-driven clause learning SAT solvers." In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), pp. 131–153.

[3] Phillip Bonacich. "Power and centrality: A family of measures." In: *American journal of sociology* (1987), pp. 1170–1182.

[4] Ulrik Brandes. "On variants of shortest-path betweenness centrality and their generic computation." In: *Social Networks* 30.2 (2008), pp. 136–145.

[5] Stephen A Cook. "The complexity of theorem-proving procedures." In: *Proceedings of the third annual ACM symposium on Theory of computing*. ACM. 1971, pp. 151–158.

[6] Glenn De'ath and Katharina E Fabricius. "Classification and regression trees: a powerful yet simple technique for ecological data analysis." In: *Ecology* 81.11 (2000), pp. 3178–3192.

[7] William F Dowling and Jean H Gallier. "Linear-time algorithms for testing the satisfiability of propositional Horn formulae." In: *The Journal of Logic Programming* 1.3 (1984), pp. 267–284.

[8] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver." In: *Theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[9] Ernesto Estrada and Juan A Rodriguez-Velazquez. "Subgraph centrality in complex networks." In: *Physical Review E* 71.5 (2005), p. 056103.

[10] Alex Flint and Matthew Blaschko. "Perceptron Learning of SAT." In: *Advances in Neural Information Processing Systems*. 2012, pp. 2771–2779.

[11] Linton C Freeman. "Centrality in social networks conceptual clarification." In: *Social networks* 1.3 (1978), pp. 215–239.

[12] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. "Satisfiability solvers." In: *Handbook of knowledge representation* 3 (2008), pp. 89–134.

[13] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring network structure, dynamics, and function using NetworkX." In: *Proceedings of the 7th Python in Science Conference (SciPy2008)*. Pasadena, CA USA, Aug. 2008, pp. 11–15.

[14] Donald E. Knuth. “Computer Programming as an Art.” In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[15] John Mingers. “An empirical comparison of pruning methods for decision tree induction.” In: *Machine learning* 4.2 (1989), pp. 227–243.

[16] David Mitchell, Bart Selman, and Hector Levesque. “Hard and easy distributions of SAT problems.” In: *AAAI*. Vol. 92. 1992, pp. 459–465.

[17] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. “Scikit-learn: Machine learning in Python.” In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[18] Vadim Ryvchin and Ofer Strichman. “Local Restarts in SAT.” In: *Theory and Applications of Satisfiability Testing*. 2008.

[19] *SAT Competition*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[20] Phil Simon. *Too Big to Ignore: The Business Case for Big Data*. Vol. 72. John Wiley & Sons, 2013.

[21] Michael Sipser. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston, 2006.

[22] *ToughSAT Generation*. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm.

[23] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. “SATzilla: portfolio-based algorithm selection for SAT.” In: *Journal of Artificial Intelligence Research* (2008), pp. 565–606.

[24] Edward N Zalta and Samson Abramsky. *Stanford encyclopedia of philosophy*. 2003.