Hide and Seek: Scaling Machine Learning for Combinatorial Optimization via the Probabilistic Method

Dimitris Achlioptas\textsuperscript{1}, Amrit Daswaney\textsuperscript{2}, and Periklis A. Papakonstantinou\textsuperscript{3}

\textsuperscript{1}University of Athens
\textsuperscript{2}Frogdata
\textsuperscript{3}Rutgers University

Abstract

Applying deep learning to solve real-life instances of hard combinatorial problems has tremendous potential. Research in this direction has focused on the Boolean satisfiability (SAT) problem, both because of its theoretical centrality and practical importance. A major roadblock faced, though, is that training sets are restricted to random formulas of size several orders of magnitude smaller than formulas of practical interest, raising serious concerns about generalization. This is because labeling random formulas of increasing size rapidly becomes intractable. By exploiting the probabilistic method in a fundamental way, we remove this roadblock entirely: we show how to generate correctly labeled random formulas of any desired size, without having to solve the underlying decision problem. Moreover, the difficulty of the classification task for the formulas produced by our generator is tunable by varying a simple scalar parameter. This opens up an entirely new level of sophistication for the machine learning methods that can be brought to bear on Satisfiability. Using our generator, we train existing state-of-the-art models for the task of predicting satisfiability on formulas with 10,000 variables. We find that they do no better than random guessing. As a first indication of what can be achieved with the new generator, we present a novel classifier that performs significantly better than random guessing (99\%) on the same datasets, for most difficulty levels. Crucially, unlike past approaches that learn based on syntactic features of a formula, our classifier performs its learning on a short prefix of a solver’s computation, an approach that we expect to be of independent interest.

\*Authors are listed in alphabetical order. Corresponding author: Amrit Daswaney (amrit-daswaney@gmail.com)
1 Introduction

Machine learning has been successfully applied to a wide range of domains where the instance distribution is not understood and thus no clear mathematical formulation of the problem can be constructed [8]. Recently, there has been great interest in investigating whether it can also be applied to combinatorial problems.

Central in this effort is Boolean Satisfiability (SAT), the canonical \textbf{NP}-complete problem [13]. Besides its immense theoretical importance, SAT has several practical applications, ranging from hardware and software verification to planning and scheduling [19]. So far, researchers have built machine learning methods to predict satisfiability [34, 10] and/or to find a satisfying assignment [34, 1).

A major roadblock in the effort to use machine learning for satisfiability is the absence of datasets comprising large labeled formulas from distributions for which the decision problem appears hard. Specifically, in order to form a (balanced) training set, all existing works generate random 3-CNF formulas “at the threshold,” so that the probability of satisfiability is (approximately) 1/2, and then use a complete (DPLL) solver to classify them, i.e., to decide whether each one is satisfiable or not. Since resolution proofs of unsatisfiability (and thus DPLL solver executions) for random formulas are exponentially large in the number of variables [12], this means that they can only label relatively small formulas, i.e., orders of magnitude smaller than practically relevant formulas. To make things worse, unlike their unsatisfiable counterparts, the satisfiable formulas generated in this manner are very easy for modern SAT solvers.

The use of formulas generated as above for training raises serious concerns about the potential generalization to formulas of practically relevant size. This is due not only to the very small size of the training formulas but also to the fragility of their generative model, as we need to be right “at the threshold” in order to get a balanced data set. Finally, a further issue is that there is good reason to suspect that the current state-of-the-art models [34, 10], having been developed for relatively small formulas, may well turn out to be impossible to train on formulas or practically interesting size.

In this work, we exploit the Probabilistic Method [3] to introduce a generator that overcomes all of the aforementioned concerns. Specifically, our generator can efficiently generate arbitrarily large, labeled random \textit{k}-CNF formulas, enabling researchers to finally train models on practical problem sizes. Our generator has the additional property that the difficulty of solving the satisfiable formulas it generates can be modulated from easy to extremely hard, i.e., apparently as hard as the unsatisfiable formulas. Finally, our generator makes it possible to consider a new learning task, wherein instead of distinguishing between satisfiable and unsatisfiable formulas, we need to distinguish between formulas that are satisfiable and formulas that are far from satisfiable, i.e., formulas where every truth assignment violates a non-trivial number of clauses.
1.1 Overview

Recall that to generate a random 3-CNF formula with \( n \) variables and \( m \) clauses one simply selects uniformly and independently \( m \) clauses out of all \( \binom{n}{3} \) possible 3-clauses on the \( n \) variables. The interesting asymptotic regime is when \( m = r n \), i.e., when \( m \) is linear in \( n \). The satisfiability threshold is conjectured to occur around density \( r \approx 4.267 \ldots \) [9].

As mentioned, the method used so far to generate training examples is the following: (1) generate a uniformly random 3-CNF formula of density \( r = 4.267 \) and a few hundred variables; (2) use a complete DPLL solver to label it, i.e., to determine whether it is satisfiable or unsatisfiable. Naturally, the second step crucially curtails the size of formulas that can be generated in this manner.

Since certifying that a given random 3-CNF formula is, typically, an intractable task, our key insight is to sidestep it completely, replacing certainty with virtual certainty, i.e., overwhelming probability. As we will see, the reason we can get away with this is that for densities not far above the satisfiability threshold, the probability that a random formula is satisfiable is exponentially small in its number of variables. As a result, our generator never invokes a satisfiability solver, proceeding instead as follows.

First, a density above the satisfiability threshold is fixed, e.g., \( r^* = 5 \). As we will see, the exact value of \( r^* \) is not very important, making the process robust and, in fact, tunable. Then, for any desired number of variables \( n \), out of all \( \binom{n}{3} \) possible clauses on the \( n \) variables, \( m = r^* n \) clauses are selected, independently, as follows:

(a) To generate a negative (unsatisfiable) example, clauses are selected uniformly.

(b) To generate a positive (satisfiable) example, first, a truth assignment \( A \in \{0, 1\}^n \) is selected uniformly. Then:

• Clauses violated by \( A \) are never selected.

• Clauses with \( t > 0 \) satisfied literals under \( A \) are selected with probability proportional to \( q^t \), where \( q < 1 \) serves as the hardness-controlling parameter.

Admittedly, a priori, it is not clear why either of the steps (a) or (b) above serves its stated purpose. In a nutshell, the idea in each case is as follows.

(a) We will see that there exists a density \( r_c \approx 4.667 \), such that if a random 3-CNF formula has density \( r = r_c + \delta \), then the probability that it is satisfiable is \( \exp(-\Theta(\delta n)) \). As a result, if, for example, we form a dataset with, say, 100,000 formulas, each having 10,000 variables and 50,000 clauses, the probability that even one of them will be satisfiable is approximately \( 10^{-148} \). While we give an explicit mathematical formula for this probability as a function of all relevant quantities, it should suffice to say that when \( r^* \geq 5 \) and \( n \geq 10,000 \), one does not really need to bother with the computation, as the probability of generating a satisfiable formula is astronomically small.

(b) Having exploited that above the threshold the probability of satisfiability is exponentially small, we need a method to generate satisfiable formulas of the same density, for, otherwise, a classifier could learn to distinguish the two classes, trivially, based on the density
alone. For this, we need to “plant” a random satisfying assignment, $A$, which is precisely what we do by never selecting clauses violated by $A$. Such naive planting, though, is not enough: if we sample the clauses satisfied by $A$ uniformly, i.e., with equal probability, the planted assignment is quite easy to find [1] and the formulas could, thus, be easily distinguished from uniformly random formulas.

To overcome the problems associated with naive planting, we leverage the work of [22] who introduced the distribution we use inspired by the work of [2]. As they have shown (and we discuss and reproduce), for each density, there is a relatively wide range of values of $q$, such that the resulting formulas are hard. The intuition is that by taking $q < 1$, the bias of each variable towards its value in $A$ is reduced, to the point that for small enough $q$ the bias is, in fact, in the opposite direction, deceiving local search solvers. Combined with the fact that the only satisfying assignments are $A$ and, perhaps, some assignments near it, hardness emerges.

Using our problem generator, we create a balanced dataset of large satisfiable and unsatisfiable formulas with 10,000 variables. On the task of predicting the satisfiability of a randomly chosen formula, existing state-of-the-art models [34, 10] do no better than random guessing. Our second contribution is to present a novel classifier that performs significantly better than random guessing (99%) on the same datasets for most difficulty levels. In contrast to existing methods that syntactically encode the formula, our classifier uses statistics of a short prefix of a solver’s computation on the formula. Concretely, we run a few independent instantiations of a popular stochastic local search algorithm WalkSAT on the input formula for a small number of steps. At every step of its computation, we record the value of various variables (e.g. the number of unsatisfied clauses and the Hamming distance between the current assignment from the starting assignment). We summarize the computation by using various summary statistics and by treating the solver’s computation as a time series and generating time-related features.

2 Prior Work

Problem generators. The machine learning community has used uniform random $k$-SAT formulas as training datasets [26, 37, 36, 34, 10]. However, in the SAT community, other formula generators are also used. The common idea is to identify some practically relevant class of formulas and use real-life instances from that class to train a model that outputs new formulas that resemble the real-life formulas. The original class can often be characterized by a small set of properties of interest. For example, researchers have found that the graph representation of several industrial formulas has a “community structure”. Informally, a graph has a community structure, if it can be decomposed into subgraphs that have more internal edges than outgoing edges. Each subgraph is called a community. The aim would then be to create new formulas that also have community structure [17].

Unfortunately, these formula generators face the same problem outlined for uniform random formulas, namely they need labeled examples in order to train the generative model, something which, in turn, requires invoking a SAT solver. This limits the size of the for-
mulas to a few thousand variables. Additionally, existing experimental evidence [5, 17, 18] suggests that these instances are easier to solve than the class of formulas they aim to resemble. In contrast, we are able to tune our generator to output formulas that range from easy to seemingly extremely hard by changing the value of \( q \). As we will see, our experiments suggest that balanced deceptive formulas, i.e., where \( q \) is chosen so that the local bias of each variable points neither towards its value in \( A \) nor towards the opposite value, is as hard as unsatisfiable formulas.

**Satisfiability prediction.** Initial work on satisfiability prediction was done as part of the development of empirical hardness models [26]. They trained a random forest classifier on the syntactic features of the formula (e.g. number of positive and negative literals) and statistics about variables in the relaxation of the problem (e.g. coefficient of variation of the slack variable in the LP relaxation of the problem) to predict the satisfiability for random 3-SAT formulas with 100, 150, \cdots, 600 variables [37, 36].

More recently, the seminal work of [34] initiated the study of satisfiability prediction using neural networks. They represent a formula using its literal-clause graph and use a graphical neural network (called ‘NeuroSAT’) with a message-passing architecture to distinguish a pair of satisfiable and unsatisfiable formulas that differ in 1 clause [34]. They consider formulas with 40 to 200 variables. There has been significant follow-up work to NeuroSAT. The most relevant of which is [10]. Using tensors to represent the formula symbolically, they apply NeuroSAT’s message-passing architecture and the exchangeable architecture of [20] to predict the satisfiability of random 3-SAT formulas with 100, 200, \cdots, 600 variables. We closely compare our work to these two prior papers. First, we use our problem generator to generate the same prediction tasks used in each work using formulas with 10,000 variables of varying difficulty levels. Using the publicly available code, we train both NeuroSAT and the architecture presented in [10] on our datasets and find that they do no better than random guessing on all difficulty levels. In contrast, we use our novel classifier on the same datasets and perform significantly better than random guessing on most difficulty levels (67% - 83%).

Additionally, there has broadly been follow-up work to NeuroSAT in 2 other directions: (a) using NeuroSAT (with or without tweaks) to augment SLS solvers [38, 39], and (b) improvements in its underlying architecture [4]. These lines of work are not directly comparable as they only evaluate their work on satisfiable formulas and [38, 4] use much smaller and easier formulas than the ones considered in this paper.

## 3 Problem Generator

In this section, we describe our problem generator that can be used to generate large random \( k \)-SAT formulas. Our problem generator has a number of desirable properties: (a) the values of \( k \), \( n \), and \( m/n \) can all be freely chosen which allows the asymptotic behavior of solvers to be studied, (b) a large number of instances can be easily and efficiently generated, and (c) the difficulty of test cases can be varied using a single parameter.

The core challenge in generating a balanced dataset of large random \( k \)-SAT formulas is obtaining provably UNSAT formulas. Generating SAT formulas are easier because finding a
single satisfying assignment is sufficient. In practice, backtracking survey propagation can be used to find satisfying assignments for large formulas very close to the phase transition [29]. In contrast, recent work in proof complexity demonstrates that refutation for random $k$-SAT requires exponential size above the phase transition (see [16] and references therein). The upshot is that there is unlikely to be an efficient algorithm for obtaining large provably UNSAT formulas.

A natural approach may be to generate instances at the phase transition and attempt to solve them. If an instance is solved, mark it as ‘SAT’, else mark it as ‘UNSAT’. This approach does not work because existing solvers don’t work at the phase transition. It is conjectured that this is due to the presence of frozen variables [29]. For a given formula, a variable is “frozen” if its assignment is the same in all satisfying assignments of the formula.

The key observation underlying our method is that formulas with clause density above the phase transition are unsatisfiable with overwhelming probability. While it is believed that this regime of exponentially small probability begins right above the threshold, this has not been established rigorously. Instead, let

$$f(r, n) = \left(\frac{7}{8}\right)^r \cdot \left(2 - \left(1 - \frac{3}{r n}\right)^m\right),$$

(1)

where $r$ is the clause density and $n$ is the number of variables. In [24] it was shown that the probability that a random 3-CNF formula with $n$ variables and $rn$ clauses is satisfiable is bounded by $f(r, n)^n$ and this is the bound we use to argue that the formulas we generate are unsatisfiable with overwhelming probability. To see this, observe that $f$ is decreasing in $r$ and crosses 1 around $r \approx 4.667$. As a result, when $r \approx 5$, we see that $f(5, n) \approx 0.966$. This means that a random 3-CNF formula with density 5 and 10,000 variables has probability of satisfiability at most $10^{-152}$.

An approach to obtain satisfiable formulas at the desired clause density is to “plant” a solution - a satisfying assignment is first chosen and subsequently, each clause is chosen based on a distribution defined by it. If this procedure is done naively by drawing satisfied $k$-SAT clauses uniformly at random from all those satisfied by the chosen assignment then the resulting instance is much easier as the majority of literals point towards the “hidden” assignment. This issue was addressed by [22] who generate a satisfiable formula with $n$ variables and $m$ clauses as follows:

1. Fix a constant $q < 1$ that modulates the “deceptiveness” of the formula.
2. Generate a random truth assignment $A \in \{0, 1\}^n$.
3. Repeat until you have $m$ clauses: pick a clause uniformly from among the $2^k \binom{n}{k}$ clauses with $k$ distinct, non-complementary literals. If the clause is satisfied by $A$ then add the clause to the formula with probability $q^t$ where $t$ is the number of literals satisfied by $A$. Else, do nothing.

The above procedure is a slight modification of the original procedure given in [22] which was outlined in the introduction. The original procedure requires us to first associate a probability with each clause and then sample each clause based on that probability. Associating
a probability with each clause becomes computationally expensive as the number of clauses increases. Instead, we first sample each clause uniformly randomly with replacement and then add it based on the probability specified by [22]. As each clause is equally likely to be picked in the first step, it is equally likely to be chosen in our amended procedure and the procedure given in [22].

For random 3-SAT, \( q^* = 0.618 \) will ensure that each variable in the formula will have no bias, i.e., they are equally likely to be true or false. By reducing the value of \( q \), we can bias the variables to point away from their assignment in \( A \). Thus, creating a “deceptive” formula.

4 Computation-based satisfiability predictor

In this section, we describe how we use features from a short prefix of an SLS solver’s computation to represent a problem instance. There are 2 steps in the process: (a) specifying the data to be collected from the solver’s computation, and (b) using summary statistics on this data.

Prior work in this area has captured a solver’s computation “externally” by recording variables the authors hypothesized were important. For example, the maximum number of variables set to true [21] and the number of steps to the best local minimum [30]. Instead of trying to invent external features that capture problem-solving progress, our key insight is to directly record the solver’s internal view of the problem. The intuition is that given the success of these solvers, these features are a more concise and useful representation of the search space. We benefit from the clever choices and pruning a solver must perform when deciding how to explore the search space.

There are 2 challenges we faced when implementing such an approach. First, selecting the solver to use. We initially decided to pick an algorithm from the SLS family of algorithms due to their success in solving random CNF formulas. We chose WalkSAT [33] due to its simplicity and because it forms the basis of several more modern SLS solvers. WalkSAT starts by randomly selecting a starting assignment and taking steps until all clauses are satisfied, or the maximum number of steps has been reached. In every step, it randomly selects an unsatisfied clause \( C \) and checks to see if any variable in \( C \) can be flipped without causing any other clause to become unsatisfied. If such a variable exists, it is selected and flipped. Otherwise, the algorithm flips either the variable in \( C \) whose flipping causes the fewest number of clauses to become violated or a random variable in \( C \). The probability of each of these two possibilities is controlled by the only parameter of the algorithm, denoted by \( wp \), modulating how greedy each step is.

Second, deciding how to represent the solver’s computation to a classifier. We choose not to give the entire computation to the classifier as this would necessarily scale with the size of the problem instance. This meant compressing the solver’s computation in some fashion. We chose to do this in two ways: (a) using summary statistics to get an overall picture of the solver’s computation, and (b) treating the solver’s computation as a time series and generating time-related features to capture the trajectory of the solver’s path. We now
describe the variables recorded of the solver’s computation and how they were transformed into features.

### 4.1 Solver’s computation-related variables

At every step of the solver’s computation, we recorded the value of the following variables:

1. The variable flipped and its break value.
2. The number of unsatisfied clauses.
3. The clause chosen.
4. The number of steps since the clause chosen was flipped.
5. Hamming distance between the current assignment and the starting assignment.

For each variable and clause, we additionally calculate their count and period.

### 4.2 Features

We obtain a list of values for each variable mentioned above at each step of its computation. Given this list as input, we generate 2 types of features:

1. **Summary statistics.** We compute the maximum, minimum, average, median, variance, coefficient of variation, first value, last value, quantile and decile values, and inter-quartile (decile) range. We included quantile and decile values to capture potentially different behavior within the same list of values.

2. **Time-series related features.** The authors in [28] systematically study time series-related features and their utility on various time series-based tasks. They propose a set of 22 features that: (a) exhibit strong classification performance across a given collection of time-series problems, and (b) are minimally redundant. We use these features to capture the solver’s trajectory. We refer the reader to page 14 of [28] for a full description of the features. In addition to these features, we also calculate the first and second derivatives of each time series.

## 5 Experiments

### 5.1 Experimental Setup

**Hardware and software.** All experiments were conducted on a compute cluster with 688 nodes, each equipped with two 2.60 GHz Intel Xeon 32-bit processors with 2 GB of RAM per processor. All experiments and analyses were written in Python 3.7. Machine learning models from Python’s scikit-learn library [31] were used. Code to replicate all of our experiments can be found [here](#).
Deceptive formula dataset. Using our problem generator described in Section 3 we create 2000 unsatisfiable random 3-SAT formulas with 10,000 variables at clause density $r \in \{5, 7, 9\}$. We create 2000 satisfiable formulas with the same number of variables, clause density, and $q \in \{0.3, 0.4, 0.5, 0.618\}$. We use a range of values for $q$ and $r$ to study their impact on classification accuracy.

Using (1), the probability that formulas we label ‘UNSAT’ are actually satisfiable is approximately $0.966^{10000} \approx 1.52 \times 10^{-153}$, $0.766^{10000} \approx 0$, and $0.595^{10000} \approx 0$ for clause densities 5, 7, and 9 respectively.

Comparison to prior work. Using the publicly available code and training procedure specified in their respective papers, we re-trained the models in [10] and [34] on the same formulas used in our experiment. In addition, for the model used in [10], we corresponded via email with the authors to ensure that we were using the correct training procedure. We trained their model for 72 hours (they use 40 hours in their paper). The model used in [34] was trained for $10^3$ iterations for formulas with 200 variables. We re-train their model for $10^6$ iterations for formulas with 10,000 variables. We find that on all datasets, their average test accuracy is no better than 50%.

Choice of SLS solver. We tested our experimental setup on WalkSAT [33] with noise parameter values $wp \in \{0.1, 0.2, \cdots, 1\}$, Schöning’s algorithm [32], ProbSAT [7], and PolyLS [27]. For ProbSAT and PolyLS, we used the parameters specified in their respective papers for 3-SAT. We obtained similar results for all models except ProbSAT with a polynomial function that used both make and break values which did no better than random guessing. We report results for WalkSAT as it gave us comparable results to ProbSAT and PolyLS and was the foundation on which the 2 models were built.

Solver parameters. For each formula with $n$ variables in the dataset, we run 16 independent instantiations of WalkSAT on the formula with noise parameter $(wp) = 0.48$ and max steps set to $2n$. We studied the effect of the number of independent trials, noise parameter value, and max steps parameters on classification accuracy. Increasing the number of independent trials increases the accuracy up to a certain value. We hypothesize that this is because the noise in the feature values due to the randomness of the algorithm decreases as we increase the number of trials. We chose the smallest value of max steps which ensures that WalkSAT can’t directly solve the formula and gave us consistent results. The performance of WalkSAT, in terms of runtime distribution, is sensitive to the value of the noise parameter [25]. Interestingly, we found that this was not the case when performance is measured in terms of classification accuracy %. We tested a range of values for the noise parameter and observed similar accuracy across a range of values and the same trend reported below - almost perfect classification accuracy ($\approx 99\%$) on $q \in \{0.3, 0.4, 0.5\}$ and no better than random guessing on $q = 0.618$. We report the findings of the best noise parameter, in terms of solving a formula, $wp = 0.48$ below.

Choice of classifier. For both tasks, we trained the following scikit-learn [31] classifiers:

1https://github.com/ChrisCameron1/End2EndSAT
2https://github.com/dselsam/neurosat
Decision Tree, Random Forest, AdaBoost, Gradient Boosting, Histogram Gradient Boosting, Multi-layer Perceptron (MLP). We used the Adam optimizer to train the MLP and tuned the learning rate. Additionally, we trained the XGBoost [11], Cat Boost [14], TabNet [6], LightGBM [23], auto-sklearn [15], and FLAML [35] classifiers. All of the classifiers had similar accuracy on all settings and the same trend reported below - almost perfect classification accuracy (≈99%) on \( q \in \{0.3, 0.4, 0.5\} \) and no better than random guessing on \( q = 0.618 \). We report the results of a decision tree classifier due to its interpretability.

**Training procedure.** For all classification tasks, we followed the procedure given below:

1. Initialize the random generator with an input seed for replicability.
2. Perform an 80-20 stratified train-test split on the dataset.
3. Train a decision tree classifier with max-depth = 3.
4. Report the accuracy % on the test set.

Repeat the above process for 100 different random seeds and report the average test accuracy %.

**Hyperparameter tuning.** We use scikit-learn’s [31] grid search optimization to tune the decision tree. For max depth, we consider the values 1, 2, ⋯, 10 and leaving it unconstrained. For max features, we consider the fractional values 0.1, 0.2, ⋯, 1, “sqrt”, “log2”, and leaving it unconstrained. For minimum samples at the leaf node, we consider the fractional values 0.1, 0.2, ⋯, 1.

**Feature Selection.** We use scikit-learn’s [31] forward recursive feature elimination with stratified 5-fold cross-validation to select the top 3 features.

### 5.2 Predicting satisfiability on individual formulas.

**Experiment.** We train our classifier on the learning task proposed in [10]. The task is to predict the satisfiability of a given random 3-SAT formula. The authors in [10] train their classifier on a balanced dataset of 5000 satisfiable and unsatisfiable formulas with 600 variables, 2,555 clauses, and clause density 4.258. We train our classifier on a balanced deceptive formula dataset with 2000 satisfiable and 2000 unsatisfiable formulas each with 10,000 variables. We vary the value of the clause density \((r)\) and deceptive parameter \((q)\) to study their impact on classification accuracy. We use the training procedure described above.

**Result.** We report the classification accuracy in Table 1. Our classifier is able to distinguish deceptive formulas \((q < 0.618)\) from uniform random unsatisfiable formulas with almost perfect classification accuracy (approximately 99%). The top features are all related to summary statistics of the number of unsatisfied clauses at every step of the computation. At a high level, the deceptiveness of the formula biases the walk away from the satisfying assignment and apparently leads to a larger number of unsatisfied clauses on average than the uniform random unsatisfiable formula. However, when the deceptive formula is balanced, i.e., where \( q \) (= 0.618 for 3-SAT) is chosen so that the local bias of each variable points
neither towards its value in the planted satisfying assignment, nor towards the opposite value
then our classifier is unable to distinguish the formula from a uniform random unsatisfiable
formula. Our classifier improves on the architecture proposed in [10] which is unable to
distinguish formulas for any values of \( q \) and \( r \). We leave open the question of whether
large balanced deceptive formulas can be \textit{efficiently} distinguished from uniform random
unsatisfiable formulas.

Prima facie, this result may seem to be at odds with the results of [1] where they report
WalkSAT taking a larger number of steps as the value of \( q \) decreases. The reason for this
apparent tension is due to the difference in the focus of the 2 papers. In [1], they focus on
making a formula hard for WalkSAT to solve. In contrast, we focus on making a satisfiable
formula hard to distinguish from an unsatisfiable formula. When \( q \) is reduced WalkSAT on
average satisfies fewer clauses for satisfiable formulas with small \( q \) than a randomly generated
UNSAT formula. We are able to use this difference to distinguish between the 2 formulas. Its
hardness to solve actually makes it easy to distinguish from a randomly generated UNSAT
formula. For example, [1] shows that WalkSAT is unable to find a satisfying assignment
for deceptive formulas with \( q = 0.2 \) even if given \( 10^6 \times n \) steps. In contrast, we show that
if we are allowed to take \textit{only} \( 2n \) steps, we are able to distinguish this formula from an
unsatisfiable formula!

\[
\begin{array}{|c|c|c|c|}
\hline
q / r & r = 5 & r = 7 & r = 9 \\
\hline
0.3 & 98.99\% (0.07\%) & 99.00\% (0.02\%) & 98.91\% (0.02\%) \\
0.4 & 99.02\% (0.06\%) & 99.03\% (0.05\%) & 99.00\% (0.03\%) \\
0.5 & 98.96\% (0.11\%) & 99.01\% (0.01\%) & 98.98\% (0.18\%) \\
0.618 & 49.90\% (1.89\%) & 49.07\% (2.09\%) & 50.04\% (2.23\%) \\
\hline
\end{array}
\]

Table 1: Classification accuracy \% of our computation-based classifier on distinguishing a
uniform random 3-SAT formula and a satisfiable formula with a deceptively planted solution
for different values of clause density (\( r \)) and deceptive parameter (\( q \)). All formulas have
10,000 variables.

5.3 Predicting satisfiability on pairs of formulas.

\textbf{Experiment.} We generalize the learning task proposed in [34]. In [34], formulas are generated by incrementally adding one clause at a time until the formula becomes unsatisfiable. A random literal of the last clause is flipped to produce a satisfying formula. The learning task is to distinguish this pair of satisfiable and unsatisfiable formulas that differ in a single clause. They consider formulas with 40 to 200 variables.

We extend their notion of a pair of formulas differing in a single clause to a pair of formulas differing in \( u \) clauses. Given a formula \( F \) with satisfying assignment \( A \), we construct a new formula \( F_u \) by uniformly randomly selecting \( u \) of \( F \)'s clauses and flipping their literals to ensure that \( A \) does not satisfy any of those clauses. We output the pair \( (F, F_u) \). The learning task is to identify the formula that \( A \) satisfies. By extending their notion, we can further ask how far apart, in terms of clauses, the formulas need to be before we can distinguish them.
We create 1000 pairs of formulas with 10,000 variables using the above process for each pair of values of $q \in \{0.3, 0.4, 0.5, 0.618\}$ and $r \in \{5, 7, 9\}$. We randomly selected 500 pairs of formulas with $F$ in the first slot and $F_u$ in the second slot and the label $y = 0$. We put $F_u$ in the first slot and $F$ in the second slot with the label $y = 1$ for the other 500 pairs of formulas. Each row contains features for the solver’s computation on each formula in the pair.

We set a threshold classification accuracy % and report the minimum fraction of clauses unsatisfied by the deceptively planted satisfying assignment ($u/rn$) before our classifier can distinguish it from the original satisfiable formula with an average test accuracy % of at least the set threshold. We start with $u = 1$ and double its value until we find a value ($U$) that crosses the threshold classification accuracy %. Once we find this value, we use binary search in the range $[U/2, U]$ to find the desired minimum value of $u$. For each value of $r$, $q$, and $u$, we follow the training procedure described above to obtain the average test accuracy %.

**Experimental Controls.** We apply the following experimental controls:

1. Fix a satisfiable formula $F$ with satisfying assignment $A$. Then for all pairs $(u, v)$ such that $u < v$, we ensure that all clauses that are not satisfied by $A$ in $F_u$ are also not satisfied in $F_v$. This is to ensure that when comparing the classification accuracy on $(F, F_u)$ and $(F, F_v)$, the only difference between the 2 pairs are the additional randomly chosen $v - u$ clauses that $A$ does not satisfy.

2. We randomly permute the variable names and order of the clauses to ensure that the classifier cannot learn the labels or order of the clauses.

3. When running WalkSAT on each formula in a pair, we ensure that they are initialized with the same random seed to control for differences due to randomness and starting assignment.

**Result.** Observe that both the satisfiable and unsatisfiable formulas in any pair of formulas are initialized with the same random seed. This means that their walks are identical for the initial few steps until they select a differing clause. Given that the length of our walk is short in comparison to the number of clauses, the finding that our classifier is able to distinguish formulas that differ in only 0.1% - 2% of their clauses is very encouraging!

The unnormalized value of $u$ is similar across different clause densities. This accounts for the observed decrease in fraction of clauses required as the deceptive parameter increases. Additionally, we observe a steep increase in the fraction of clauses required for the balanced deceptive formulas. This strengthens our finding from Experiment 1 that balanced deceptive formulas are hard to distinguish.
Figure 1: The minimum % of clauses unsatisfied by the deceptively planted satisfying assignment before our classifier can distinguish it from the original satisfiable formula with an average test accuracy % of at least 70%.

6 Discussion

Other problems. The only barrier to applying our approach to other problems (e.g. graph colorability, expressed as a satisfiability problem as done in [34]) is obtaining correctly labeled formulas of practical size.

Industrial instances. We acknowledge that the instances created by our problem generator lack the structure of industrial instances that practical solvers exploit to obtain satisfying assignments. However, uniform random $k$-SAT instances have a long history of being instrumental in the discovery of insights and development of solvers that were later utilized in industrial solvers. We expect our problem generator to help in a similar fashion. Moreover, given the difficulty of our instances, we expect them to provide useful lower bounds on the performance of solvers.

Meaningful in the breach. This work was motivated, in part, to test the tacit premise of most current work in this area that performance on smaller formulas is indicative of performance on larger formulas. By modulating the parameter $q$ in our problem generator,
one can obtain large datasets of varying difficulty. While success on our dataset may not
guarantee that your model’s performance generalizes to large formulas, we posit that failure
to do so may provide a strong reason to evaluate your model’s performance.

**Recovering satisfying assignments.** A limitation of our work is that there is no im-
mediate way to recover satisfying assignments when the classifier predicts ‘SAT’. Future
work could involve using our predictor as an oracle for CDCL-based solvers for early conflict
prediction and variable selection.

**Theoretical questions.** While our work provides evidence that computation-based solvers
outperform syntactic-based solvers on predicting satisfiability, a theoretical investigation as
to why this occurs is still open. A direction we find particularly interesting is the connection
to interactive proofs (IP). When interactive proofs were first introduced, it wasn’t immedi-
ately clear that they were going to be (much) more powerful than conventional proofs in \( \text{NP} \)^3. However, we now know that allowing the prover and verifier to communicate is very power-
ful. Indeed, IP can decide any language in \( \text{PSPACE} \) with arbitrarily small error. If we recast
SAT solvers as provers, our work suggests that incomplete proofs could still contain mean-
ingful information. This observation has already been made in the study of zero-knowledge
proofs when dealing with cheating verifiers that prematurely end the conversation. If this is
indeed true, then this raises questions about whether this information can be algorithmically
obtained and used to improve performance.

## References

[1] Dimitris Achlioptas, Haixia Jia, and Cristopher Moore. “Hiding satisfying assignments:
two are better than one”. In: *Journal of Artificial Intelligence Research* 24.1 (Nov.
2005), pp. 623–639.

[2] Dimitris Achlioptas and Yuval Peres. “The threshold for random k-SAT is 2k (ln
2 - O(k))”. In: *Proceedings of the thirty-fifth annual ACM symposium on Theory of
computing*. STOC ’03. New York, NY, USA: Association for Computing Machinery,
June 2003, pp. 223–231.

[3] Noga Alon and Joel Spencer. *The Probabilistic Method*. John Wiley, 1992.

[4] Saeed Amizadeh, Sergiy Matusievych, and Markus Weimer. “Learning To Solve Circuit-
SAT: An Unsupervised Differentiable Approach”. In: *7th International Conference on
Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*
Open-Review.net, 2019.

[5] Carlos Ansótegui, Maria Luisa Bonet, and Jordi Levy. “Towards industrial-like random
SAT instances”. In: *Proceedings of the 21st International Joint Conference on Artificial
Intelligence*. IJCAI’09. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.,
July 2009, pp. 387–392.

^3https://courses.cs.washington.edu/courses/cse533/05au/pcp-history.pdf
[6] Sercan O Arik and Tomas Pfister. “TabNet: Attentive Interpretable Tabular Learning”. In: Thirty-Fifth AAAI Conference on Artificial Intelligence, AAAI 2021, Thirty-Third Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The Eleventh Symposium on Educational Advances in Artificial Intelligence, EAAI 2021, Virtual Event, February 2-9, 2021. AAAI Press, 2021, pp. 6679–6687.

[7] Adrian Balint and Uwe Schöning. “Choosing probability distributions for stochastic local search and the role of make versus break”. In: Proceedings of the 15th international conference on Theory and Applications of Satisfiability Testing. SAT ’12. Berlin, Heidelberg: Springer-Verlag, June 2012, pp. 16–29.

[8] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. “Machine learning for combinatorial optimization: A methodological tour d’horizon”. In: Eur. J. Oper. Res. 290.2 (2021), pp. 405–421.

[9] A. Braunstein, M. Mézard, and R. Zecchina. “Survey propagation: An algorithm for satisfiability”. en. In: Random Structures & Algorithms 27.2 (2005). eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.20057, pp. 201–226.

[10] Chris Cameron, Rex Chen, Jason Hartford, and Kevin Leyton-Brown. “Predicting Propositional Satisfiability via End-to-End Learning”. en. In: Proceedings of the AAAI Conference on Artificial Intelligence 34.04 (Apr. 2020). Number: 04, pp. 3324–3331.

[11] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016. Ed. by Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi. ACM, 2016, pp. 785–794.

[12] Vašek Chvátal and Endre Szemerédi. “Many hard examples for resolution”. In: Journal of the ACM 35.4 (Oct. 1988), pp. 759–768.

[13] Stephen A. Cook. “The complexity of theorem-proving procedures”. In: Proceedings of the third annual ACM symposium on Theory of computing. STOC ’71. New York, NY, USA: Association for Computing Machinery, May 1971, pp. 151–158.

[14] Anna Veronika Dorogush, Vasily Ershov, and Andrey Gulin. “CatBoost: gradient boosting with categorical features support”. In: CoRR abs/1810.11363 (2018). arXiv: 1810.11363.

[15] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. “Efficient and Robust Automated Machine Learning”. In: Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada. Ed. by Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett. 2015, pp. 2962–2970.

[16] Noah Fleming, Denis Pankratov, Toniann Pitassi, and Robert Robere. “Random (log n) -CNFs are Hard for Cutting Planes”. In: Journal of the ACM 69.3 (June 2022), 19:1–19:32.
[17] Jesús Giráldez-Cru and Jordi Levy. “A Modularity-Based Random SAT Instances Generator”. In: *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*. Ed. by Qiang Yang and Michael J. Wooldridge. AAAI Press, 2015, pp. 1952–1958.

[18] Jesús Giráldez-Cru and Jordi Levy. “Locality in Random SAT Instances”. In: *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*. Ed. by Carles Sierra. ijcai.org, 2017, pp. 638–644.

[19] Carla P. Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. “Chapter 2 Satisfiability Solvers”. In: *Foundations of Artificial Intelligence*. Ed. by Frank van Harmelen, Vladimir Lifschitz, and Bruce Porter. Vol. 3. Handbook of Knowledge Representation. Elsevier, Jan. 2008, pp. 89–134.

[20] Jason Hartford, Devon Graham, Kevin Leyton-Brown, and Siamak Ravanbakhsh. “Deep Models of Interactions Across Sets”. In: *Proceedings of the 35th International Conference on Machine Learning*. ISSN: 2640-3498. PMLR, July 2018, pp. 1909–1918.

[21] Eric Horvitz, Yongshao Ruan, Carla Gomes, Henry Kautz, Bart Selman, and Max Chickering. “A bayesian approach to tackling hard computational problems”. In: *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*. UAI’01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Aug. 2001, pp. 235–244.

[22] Haixia Jia, Cristopher Moore, and Doug Strain. “Generating hard satisfiable formulas by hiding solutions deceptively”. In: *Proceedings of the 20th national conference on Artificial intelligence - Volume 1*. AAAI’05. Pittsburgh, Pennsylvania: AAAI Press, July 2005, pp. 384–389.

[23] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. “LightGBM: A Highly Efficient Gradient Boosting Decision Tree”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*. Ed. by Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett. 2017, pp. 3146–3154.

[24] Lefteris M. Kirousis, Evangelos Kranakis, Danny Krizanc, and Yannis C. Stamatiou. “Approximating the unsatisfiability threshold of random formulas”. In: *Random Struct. Algorithms* 12.3 (1998), pp. 253–269.

[25] Lukas Kroc, Ashish Sabharwal, and Bart Selman. “An Empirical Study of Optimal Noise and Runtime Distributions in Local Search”. In: *Theory and Applications of Satisfiability Testing – SAT 2010*. Ed. by Ofer Strichman and Stefan Szeider. Berlin, Heidelberg: Springer, 2010, pp. 346–351.

[26] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. “Empirical hardness models: Methodology and a case study on combinatorial auctions”. In: *Journal of the ACM* 56.4 (July 2009), 22:1–22:52.
[27] Sixue Liu and Periklis Papakonstantinou. “Local Search for Hard SAT Formulas: The Strength of the Polynomial Law”. en. In: Proceedings of the AAAI Conference on Artificial Intelligence 30.1 (Feb. 2016). Number: 1.

[28] Carl H. Lubba, Sarab S. Sethi, Philip Knaute, Simon R. Schultz, Ben D. Fulcher, and Nick S. Jones. “catch22: CAnonical Time-series CHaracteristics”. en. In: Data Mining and Knowledge Discovery 33.6 (Nov. 2019), pp. 1821–1852.

[29] Raffaele Marino, Giorgio Parisi, and Federico Ricci-Tersenghi. “The backtracking survey propagation algorithm for solving random K-SAT problems”. en. In: Nature Communications 7.1 (Oct. 2016). Number: 1 Publisher: Nature Publishing Group, p. 12996.

[30] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. “Understanding random SAT: beyond the clauses-to-variables ratio”. In: Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming. CP’04. Berlin, Heidelberg: Springer-Verlag, Sept. 2004, pp. 438–452.

[31] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake VanderPlas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Mathieu Perrot, and Édouard Duchesnay. “scikit-learn: Machine Learning in Python”. In: J. Mach. Learn. Res. 12 (2011), pp. 2825–2830.

[32] T. Schoning. “A probabilistic algorithm for k-SAT and constraint satisfaction problems”. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039). ISSN: 0272-5428. Oct. 1999, pp. 410–414.

[33] Bart Selman, Henry A. Kautz, and Bram Cohen. “Local Search Strategies for Satisfiability Testing”. In: DIMACS Series in Discrete Mathematics and Theoretical Computer Science. 1996, pp. 521–532.

[34] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. “Learning a SAT Solver from Single-Bit Supervision”. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net, 2019.

[35] Chi Wang, Qingyun Wu, Markus Weimer, and Erkang Zhu. “FLAML: A Fast and Lightweight AutoML Library”. In: Proceedings of Machine Learning and Systems 2021, MLSys 2021, virtual, April 5-9, 2021. Ed. by Alex Smola, Alex Dimakis, and Ion Stoica. mlsys.org, 2021.

[36] Lin Xu, Holger Hoos, and Kevin Leyton-Brown. “Predicting Satisfiability at the Phase Transition”. en. In: Proceedings of the AAAI Conference on Artificial Intelligence 26.1 (2012). Number: 1, pp. 584–590.

[37] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. “Hierarchical hardness models for SAT”. In: Proceedings of the 13th international conference on Principles and practice of constraint programming. CP’07. Berlin, Heidelberg: Springer-Verlag, Sept. 2007, pp. 696–711.
[38] Emre Yolcu and Barnabas Poczos. “Learning Local Search Heuristics for Boolean Satisfiability”. In: Advances in Neural Information Processing Systems. Vol. 32. Curran Associates, Inc., 2019.

[39] Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. “NLocalSAT: Boosting Local Search with Solution Prediction”. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 1177–1183.