An Algorithm Selection Approach for QF_FP Solvers

Joseph Scott, Pascal Poupart, and Vijay Ganesh

University of Waterloo, Canada
{joseph.scott,ppoupart,vganesh}@uwaterloo.ca

Abstract

Floating-point numbers are a fundamental data type used in numerous safety-critical software systems. Due to their popularity, verification of programs consisting of floating-point calculations is in high demand. Several modern Satisfiability Modulo Theory (SMT) solvers, such as Z3, CVC4, MathSAT5, now support theories of first-order floating-point arithmetic. Modern floating-point solvers implement a variety of algorithms to test satisfiability of floating-point equations, such as '*eager*' bit-blasting or other '*lazy*' approaches. Due to the variety of floating-point SMT algorithms, it is very natural to ask which floating-point SMT algorithm to use for a particular floating-point equation.

In this paper, we initiate solving this problem by an algorithm selection or an algorithm '*portfolio*' over floating-point SMT solvers. Specifically, we consider a variety of supervised machine learning algorithms to classify and regress runtimes of floating-point SMT inputs to an array of solvers. We do analysis our generated inputs as well as input benchmarks from the SMT-COMP '18.

1 Introduction

Floating-point arithmetic is highly utilized and optimized in modern computer systems. Reports of modern CPUs capable of hundreds of billions of floating-point arithmetic operations per second (FLOPs), while modern GPUs are capable of performing hundreds of trillions of FLOPs [27]. Consequently, scientific software, often with safety-critical components, will leverage the speed of floating-point in their calculations. Applying verification could potentially have a significant impact on making floating-point software more robust and dependable.

Modern software verification tools often depend on internal calls to SMT solvers during the verification process. In the case of floating-point programs, verification tools will query satisfiability of first-order equations of floating-point arithmetic.

The algorithms used to solve floating-point SMT come in various forms. The SMT solvers Z3 and CVC4 implement eager bit-blasting, by reducing the floating-point arithmetic to its circuit and solving an instance of a bit-vector SMT problem [11, 2, 7]. Other solvers such as MathSAT5 and Colibri, implement 'lazy' SMT algorithms. MathSAT5 includes a lattice-theoretic interval propagation approach to deciding satisfiability of floating-point equations with their algorithm: Abstract Conflict Driven Clause Learning (ACDCL). Colibri, similar to MathSAT5, uses interval and difference-bound matrices (DBM) propagation with additional features, such as real linearization and applying simplex solvers.

In this paper, we implement an algorithm selection routine or algorithm '*portfolio*' of the above mentioned QF_FP SMT solvers. We use supervised learning methods such as linear ridge regression, support vector machines, and random forests. We use features of the counts of terms, functions, and predicates in the theory of floating-point arithmetic to train a classifier to SMT-LIB input to a solver.

Contributions

In this paper we make the following contributions:

1. We present, to the best of our knowledge, the first formulation of algorithm selection or algorithm portfolio of floating-point SMT. Further, to the best of our knowledge, we believe this is the first such formulation for any SMT-LIB theory.

2. We provide an empirical evaluation of our approach on benchmarks from SMT-COMP '18 over QF_FP.

3. We introduce a new dataset over QF_FP to further diversify the analysis of solver performance.

The remainder of this paper is organized as follows: Section 2 provides the necessary background for this paper by discussing floating-point and supervised learning. Section 3 goes over the necessary details of the implementation, solvers considered, and datasets considered. Section 4 goes over the experimentation of our approach. Section 5 goes over the related literature of this paper, specifically details on SatZilla [30, 31, 32, 33, 9]. Finally, Section 6 concludes the paper.

2 Background

In this section, we will go over the necessary background for this paper.

2.1 Floating Point Arithmetic

Floating-Point (FP) numbers are a fundamental datatype used frequently in computer programs. Internally, FP numbers are composed of a sign bit, and two bit-vectors representing a mantissa, and an exponent. The real number corresponding to the FP number can be interpreted as:

$$(-1)^s(1.m_1m_2m_3..m_n)_2 \times 2^{e-bias}$$

where s is the sign bit, m_i are the bits in the mantissa bit-vector, e is the unsigned integer value of the exponent bit-vector, and *bias* is a fixed integer value which is determined based on the width of the FP number. The leading 1 before the decimal point is implicit and not stored in the data structure. The subnormal case is when the unsigned integer value $e = 0$, and the previously implicit 1 becomes an implicit 0. The sign bit allows for both ± 0. If the exponent bit vector is composed entirely of ones and the mantissa is composed entirely of zeros, then the FP number denotes $\pm\infty$ depending on the sign bit. Furthermore, if the exponent is all ones and the logical or over all the bits in the mantissa is true, then the FP number denotes Not a Number (NaN). For portability, the IEEE 754 standard was introduced in 1985 and revised again in 2008 to provide technical specifications of the behavior of FP arithmetic. The IEEE 754 standard defines several familiar FP arithmetic operators, such as: *absolute value, negation, addition, subtraction, multiplication, division, fused multiply-add, and square root.* The IEEE 754 standard requires these operators within the standard to be rounded *infinitely precise*: the real value that would result of the operand from the FP inputs needs to be rounded to the closest FP number with respect to a rounding mode. The IEEE 754 requires equality as well as other inequalities to respect the real value semantics [20].

Floating-point numbers are convenient for computer programmers due to their speed compared to arbitrary precision numbers. However, FP numbers are unintuitive and frequently misunderstood, causing surprising and hard to find bugs in software. The unintuitive behavior arises from the differences between FP and the numbers they attempt to represent—the reals.

Name	Description
N	Total number of occurrences of terms in the input (constants, variables, operators, predicates, assertions.)
N_c	Total number of constants
N_v	Total number of variables
N_{op}	Total number of operators
N_{pred}	Total number of predicates
N_{assert}	Total number of assertions
$32-bit?$	If input contains at least one 32-bit float: 1.0, otherwise: 0.0
$64-bit?$	If input contains at least one 64 bit float: 1.0, otherwise: 0.0
$128-bit?$	If input contains at least one 128 bit float: 1.0, otherwise: 0.0
Variant	If the input contains at least one float that is not 32-bit, 64-bit, or 128-bit: 1.0, otherwise 0.0
$fp.abs\%$	$N_{fp.abs}/N_{ops}$, the percentage of $fp.abs$ over the total number of operands
$fp.neg\%$	$N_{fp.neg}/N_{ops}$, the percentage of $fp.neg$ over the total number of operands
$fp.add\%$	$N_{fp.add}/N_{ops}$, the percentage of $fp.add$ over the total number of operands
$fp.mul\%$	$N_{fp.mul}/N_{ops}$, the percentage of $fp.mul$ over the total number of operands
...	...
$fp.eq\%$	$N_{fp.eq}/N_{pred}$, the percentage of $fp.eq$ over the total number of predicates
$fp.lt\%$	$N_{fp.lt}/N_{pred}$, the percentage of $fp.lt$ over the total number of predicates
...	...

Table 1: Table of Features used in all experiments

For example, not all real numbers can be represented by FP numbers (e.g., 0.1 has no finite representation in any length binary FP format) so FP arithmetic must incorporate rounding; FP addition and multiplication are not associative, and the pair do not distribute; FP arithmetic can overflow or underflow; and FP has several special constants: $\pm 0, \pm\infty, NaN$. Furthermore, from a performance perspective, FP arithmetic can be inconsistent despite being of fixed length: subnormal FP numbers have been reported to slow computation by up to 100 times for specific architectures [12, 13].

2.2 Supervised Learning

Supervised learning is a branch of machine learning where a dataset of features is provided with labels. The task of a supervised learning algorithm is either regression: learning a function $f : X \to \mathbb{R}$; or a classifier: learning a function $f : X \to \mathcal{C}$; where X is the set of features describing an input and $\mathcal{C}$ is a finite set of classes over X. The dataset X is split into two, a training set and a testing set. The supervised learning algorithm either performs regression over the training set and its labels or learns a classifier. Commonly, a regression algorithm can be tweaked in its theoretical formulation to be a classification algorithm and vice-versa.

We will briefly enumerate the considered learning algorithms in this paper:

- Linear Regression - learns a linear polynomial with an objective function of minimizing the mean square error over the training set.
- Linear Ridge Regression - is an extension to Linear Regression that adds the norm of coefficients of the learned polynomial to the objective function.

- Support Vector Machines (SVM) - Support Vector Machines is a classifier (with a regression formulation SVR) that learns a hyperplane to separate classes such that the margin between points and the hyperplane is maximized.

- (k) Nearest Neighbors - is a classification algorithm (with regression formulations) that makes classification decisions by sampling the k closest points of the training set.

- Logistic Regression - A classification algorithm that infers the probability of membership of a class given the features.

- Linear Perceptron - A biologically inspired classifier that learns a linear hyper-plane that separates two classes. This can be generalized to multi-class by training one class against all for each considered class.

- Random Forests - Uses an ensemble learning approach over a 'forest' of several decision trees. Each decision tree votes on a class or regressed value and is propagated up to a final decision.

- Neural Networks - A biologically inspired algorithm that emulates a directed acyclic graph of neurons firing messages to one and another.

For further detail we refer to: [26, 19, 21, 25, 4].

3 Solvers, Implementation, and datasets

In this section, we will discuss the necessary implementation details, solvers considered, and datasets used.

3.1 Solvers

We will exclusively consider the following list of solvers:

1. **Z3** v4.8.0 - A multi-theory open source SMT solver by Microsoft Research [11]. Z3 implements FP SMT by a reduction to arithmetic over bit-vectors for each FP operator.

2. **MathSAT5** v5.5.3. A multi-theory SMT solver from FBK-IRST and DISI-UniTN [10]. MathSAT5 implements an Abstract Conflict Clause Driven Learning (ACDCL) [6] algorithm for their FP solver. MathSAT5 additionally provides bit-blasting approaches, but in this paper we only consider the ACDCL configuration.

3. **CVC4** v1.7 - A multi theory open source SMT Solver by Stanford [2]. CVC4 implements FP SMT similarly to Z3 by bit blasting FPU circuits [7].

4. **Colibri** v2070 - A proprietary CP Solver with specialty in FP SMT developed by CEA LIST. [5, 18]

We use a global timeout of 2500 seconds, a time slightly above the cutoff of the 2019 SMT-COMP. If a solver has a runtime error of any kind or fails to return a 'SAT' or 'UNSAT' result, the solver-input pair is labeled as a timeout.

3.2 Implementation

We implement our algorithm selection framework as a python package. We use SkLearn [23, 8] as our machine learning backend.

Features: Supervised machine learning is often only as powerful as its dataset and features used. Unfortunately, coming up with a proper set of features that captures the semantics of an input for the theory of QF_FP is very challenging. In fact, in our work, we will use basic features, composed of counts and ratios over the terms of the formula. Furthermore, Table 1 enumerates the considered features of all experiments.

Algorithm Selection as a Classification Problem: Algorithm selection has a straightforward reduction to classification. For each input, the features from Table 1 are computed. For every input within the considered datasets, each considered solver solves the input and times are recorded. The computed feature vector is then labeled as the solver with the shortest runtime. Inputs that no solver solves within the time limit are discarded.

Algorithm Selection as a Regression Problem: We train a regressor model for each considered solver using the features from Table 1, with labels of the log runtime. On inputs that timeout the runtime is labeled as triple the timeout. Selecting a solver is done by: predicting the log runtime of all solvers and computing the argmax. This methodology is inspired by SatZilla[33].

Training and Evaluation: For each considered dataset or combinations thereof, we partition into two sets with 50% of all data going into a training set and 50% into a testing set. Training set features are scaled to zero mean and unit deviance. The testing set is then scaled to the mean and deviance of the training set. Depending on the considered learning algorithm, 20% of the training set may be used as a validation dataset to determine the hyperparameters of the algorithm. Upon conclusion of the validation phase, the model is then retrained with the returned hyperparameters from validation over the entire training set. Throughout the training process, the learning algorithm does not have access to the testing set.

3.3 Datasets

We will consider two datasets. First, the 40,300 inputs from the 2018 SMT-COMP on QF_FP. While 40,300 inputs is a great deal of data, over 99.2% of the inputs are solved by all the solvers within 3 seconds, as they are intended as unit tests for the solvers. For the second dataset, we use a randomly generated dataset over QF_FP. This dataset is intended to provide more diversity in experiments and input difficulty than what is currently available.

Random QF_FP SMT Generation procedure: The input will be comprised of entirely 32-bit floats or 64-bit floats and is selected at the start uniformly at random. The number of variables is chosen uniformly at random in the interval $[1, 20]$. The input consists of $[1, 20]$ assertions with each asserting an Abstract Syntax Tree (AST) over floating-point arithmetic. The root node consists of one of the predicates selected uniformly at random. Depending on the arity of the selected predicate, the required number subtrees are generated with roots of floating-point operators chosen uniformly at random. This process is repeated for a fixed net depth of $[2, 6]$ chosen uniformly at random to which floating point variables selected uniformly at random fill out the leaf nodes of the AST.

4 Empirical Analysis

In this section, we present an empirical analysis of our algorithm selection framework. More specifically, we try to answer the following research question:

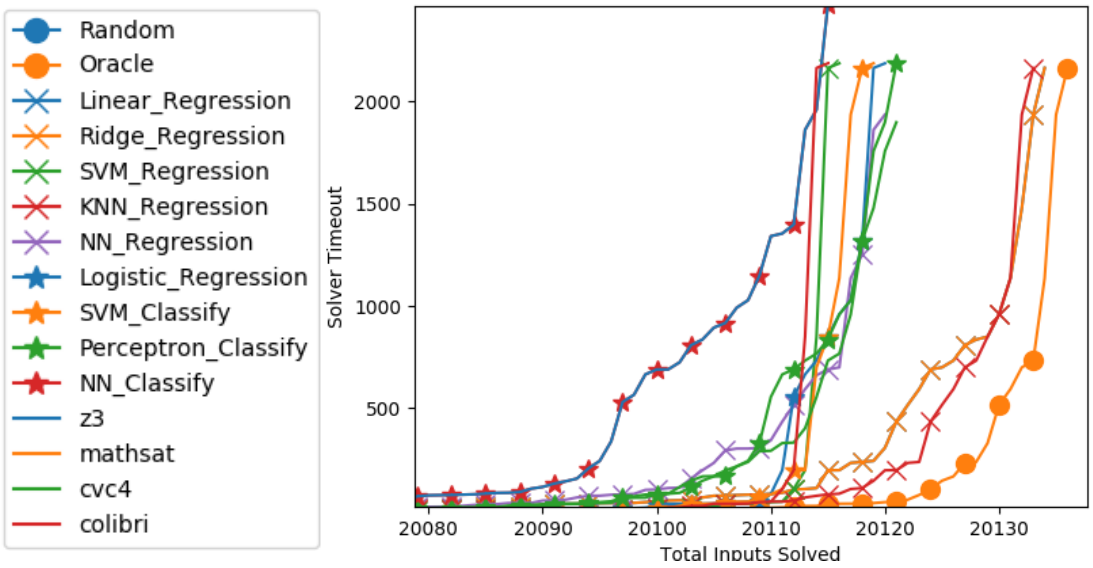

Figure 1: Cactus Plot of Experiment #1

Research Question: *Can we use machine learning and algorithm selection to improve over any standalone SMT Solver for QF_FP? How close does our proposed, yet minimalistic, feature set get to perfect algorithm selection for our benchmarks?*

4.1 Evaluation Techniques

We will use two types of figures to visualize our results

1. Cactus Plot – Demonstrates a solvers performance over a set of benchmarks, with the X-axis denoting the total number of solved inputs and the Y-axis denoting the solver timeout in seconds. A point (X, Y) on a cactus plot can be interpreted as: the denoted solver can solve X of the inputs from the benchmark set where each input has a timeout of Y seconds.
2. Confusion Matrix – A matrix that tallies all predictions of a classifier. Each row is labeled as a solver, denoting that the learning algorithm predicted the corresponding solver, and each column labeled as a solver, denoting the solver that solved the input the fastest. Furthermore, the number in the matrix $M[s_1][s_2]$ is the tally of how often the machine learning algorithm predicted solver s_1 would have the fastest running time while solver s_2 had the fastest running time. A perfect confusion matrix would be entirely zeros except for the main diagonal.

In all cactus plots we will further consider the following baselines to compare against:

1. Each solver individually
2. Random Solver Selection
3. An Oracle – Perfect Algorithm Selection, also referred to as the *virtual best solver* in various literature

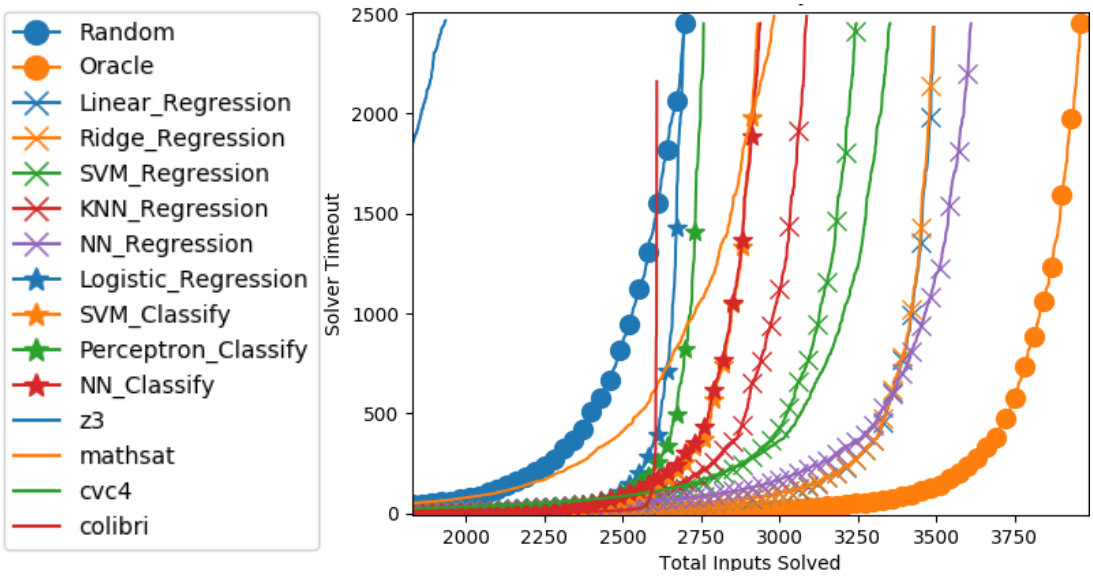

Figure 2: Cactus Plot of Experiment #2

Furthermore, for plotting clarity, we use unmarked solid lines to denote original solvers, marked lines with circles to indicate Random Selection or Oracle Selection, marked lines with crosses to denote regression based algorithms, and marked lines with stars to denote classification based algorithms.

A learned algorithm selection tool would ideally improve on all considered solvers and random selection. Further, it additionally should be competitive with an Oracle.

All experiments were performed on a CentOS V7 cluster of Intel Xeon Processor E5-2683 at 2.10 GHz. Each run of a solver was configured to be restricted to 8GB without parallelization. Otherwise, solvers were run as close to their default configurations as possible. We observe prediction times to take a few milliseconds at most and are not included in timing analysis.

Lines in cactus plots may be cropped off for readability of figures. Plot lines will only be cropped off if the solver/algorithm selection tools are not 'competitive' and trails behind by several thousand inputs.

4.2 Experiment #1: Training and Testing over the SMT-COMP 2018 dataset

We will first consider the 40,300 inputs from the SMT-COMP in 2018. We first randomly partition the dataset into 20,150 inputs for training and 20,150 inputs for testing. We want first to make note that despite the dataset being relatively large, 99% of the inputs are solved within 3 seconds. We use the training method described in Section 3.2.

Figure 1 presents the Cactus plot from this experiment. We observe that most learned algorithm selection models outperform the single solver benchmarks, and all outperform the random benchmark. We find that Linear Ridge Regression is the most competitive with the Oracle selector. This is consistent with [33], who exclusively used ridge regression in practice.[1]

[1]NN Classify picks Z3 every time!

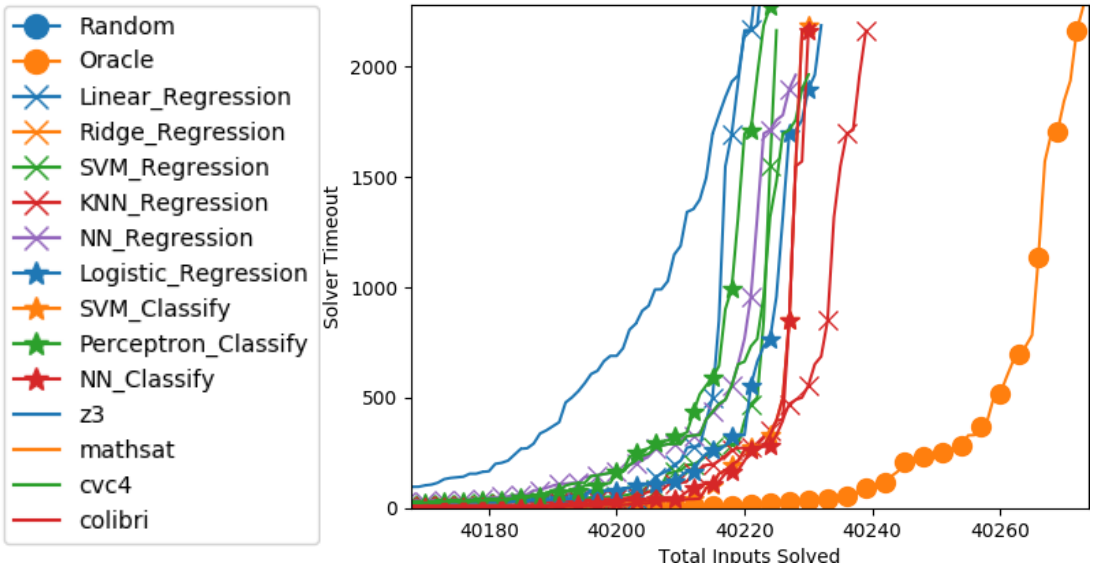

Figure 3: Cactus Plot of Experiment #3

Further, there is a notably low margin between the best performing learned algorithm selection model and the Oracle. Within this dataset, the oracle is only able to solve ten additional instances. This margin will increase in later experiments.

Figure 5 (top left) presents the confusion matrix of linear ridge regression. The confusion matrix is unsurprising due to the strong performance of Z3 on this dataset and is by far the best in terms of accuracy across all experiments.

4.3 Experiment #2: Training and Testing over the Randomly Generated dataset

We will next consider the 10,000 [2] inputs from the randomly generated dataset. We form a training set of 3,951 inputs and a testing set of 3,952 inputs. We use the training method described in Section 3.2.

Figure 2 presents the Cactus plot from this experiment. We observe that most learned algorithm selection models outperform the single solver benchmarks, and all outperform the random benchmark. We observe the Neural Network with Regression, closely followed by Linear Regression and Linear Ridge Regression are the most performant. This further remains consistent with [33] and the use of linear ridge regression.

The margin of improvement between the best algorithm selection formulation and the Oracle is much larger than observed in Section 4.2. There are more than 250 additional inputs that were solved by the oracle.

Figure 5 (top right) presents the confusion matrix for the neural network. The confusion matrix is the worst across all experiments and suggests poor classification as the higher tallies do not necessarily fall on the main diagonal, despite a successful looking cactus plot. This further motivates for the advancement of algorithm selection on more diverse datasets such as this.

[2] 2,095 were discarded as no considered solver was able to solve them within the 2,500 second timeout.

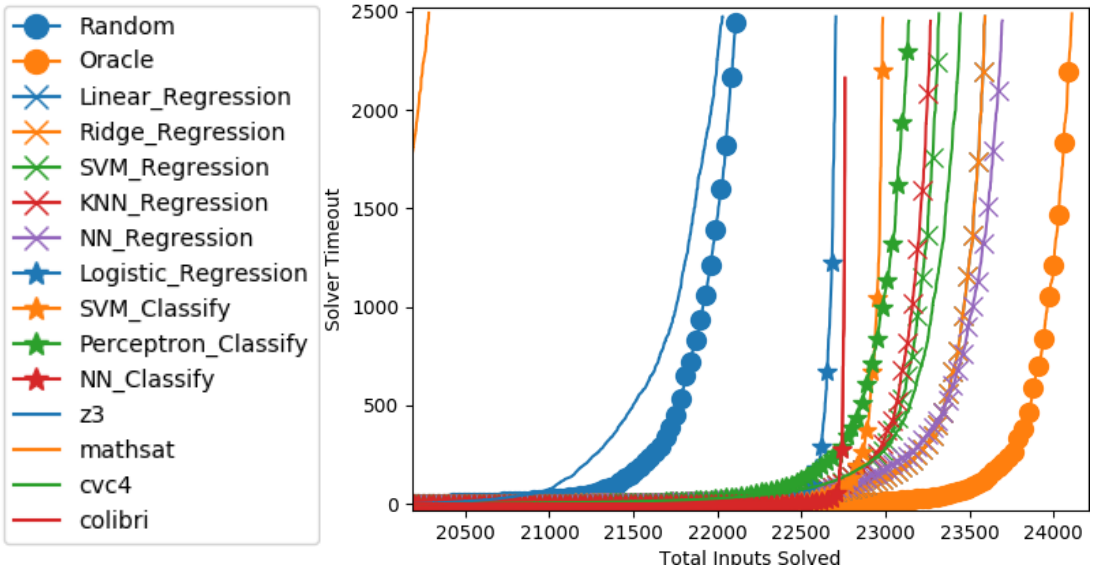

Figure 4: Cactus Plot of Experiment #4

4.4 Experiment #3: Training over the Randomly Generated Dataset, Testing over the SMT-COMP 2018 dataset

In this experiment, we train on one domain of QF_FP problem class and evaluate the learned models on another. We train on the entire 7,905 inputs from the random dataset and evaluate the learned models over the entire 40,300 inputs from the SMT COMP '18 dataset.

Figure 3 presents the Cactus plot from this experiment. We observe that most learned algorithm selection models outperform the single solver benchmarks, and most outperform the random benchmark. We observe a fairly surprising result that $kNN-$ regression outperforms all learned algorithm selection models, which is inconsistent with [33].

Figure 5 (bottom left) presents the confusion matrix for $kNN-$ regression. The confusion matrix looks much more successful than the previous experiment in Section 4.3, with the majority of the tallies falling on the main diagonal.

The Inverse of this Experiment: Training over SMT-COMP '18 dataset and testing over the Randomly Generated dataset does not result in a learned algorithm selection that beats random algorithm selection nor any standalone solver. This is not overly surprising as in SMT-COMP '18 there is a heavy favor in Z3 and learned models almost exclusively predict Z3, despite it being the worst standalone solver over that dataset.

4.5 Experiment #4: Train over both, Test over both

We combine the two considered datasets, to form a cumulative dataset of 50,300 inputs. We split each by 50% to compose the training and testing set. That is, 20,150 from SMT-COMP '18 and 3,951 from the Random Set to form a training set, and the remaining 20,150 from SMT-COMP '18 and 3,952 from the Random Set to form the testing set.

Figure 4 presents the Cactus plot from this experiment. We observe that most learned algorithm selection models outperform the single solver benchmarks, and all outperform the

	Z3	MathSAT5	CVC4	Colibri
Z3	20009	13	0	6
MathSAT5	4	8	2	36
CVC4	8	4	6	16
Colibri	1	4	2	30

	Z3	MathSAT5	CVC4	Colibri
Z3	7	8	21	32
MathSAT5	14	142	205	479
CVC4	57	445	599	1007
Colibri	24	133	169	610

	Z3	MathSAT5	CVC4	Colibri
Z3	34365	15	2	0
MathSAT5	5	9	1	22
CVC4	2878	10	12	33
Colibri	2801	30	17	100

	Z3	MathSAT5	CVC4	Colibri
Z3	19999	62	38	77
MathSAT5	51	339	265	812
CVC4	31	168	536	998
Colibri	39	36	822	628

Figure 5: Confusion Matrix of top performing learning algorithm for Experiment #1 (top left), #2 (top right), #3 (bottom left), and #4 (bottom right)

random benchmark. We observe the Neural Network with Regression, closely followed by Linear Regression and Linear Ridge Regression are the most performant. This further remains consistent with [33] and the use of linear ridge regression.

Figure 5 (bottom right) presents the confusion matrix for the learned neural network. The confusion matrix looks much more successful, similar to Section 4.4 than the experiment in Section 4.3, with the majority of the tallies falling on the main diagonal.

5 Related Work

There have been several works of algorithm selection in several domains, such as SAT solvers by SatZilla [30, 31, 32, 33, 9]. SatZilla has a vast literature and a great deal of success in the SAT competition [22]. The success of SatZilla is the primary inspiration for this paper. To the best of our knowledge, we are the first to consider algorithm selection for QF_FP or any other SMT theory. SatZilla uses a diverse feature set over propositional SAT. Their features include metrics over variables clause graphs, horn clauses, solver probing, etc. In contrast, our formulation leaves a lot to be desired, but unclear to us on how to properly expand it.

Alternatively to propositional SAT, algorithm selection has been considered in other domains [3], such as quantified boolean formulas [24, 17], Constraint Solving [14, 1, 16], answer set programming [15], and other $NP-$Hard problems [28, 29]

6 Conclusions and Future Work

In this paper, we presented an algorithm section or algorithm portfolio styled approach to floating-point SMT. We considered a variety of classification and regression algorithms used in supervised machine learning. We discovered, that with even straightforward features, an algorithm selection approach can outperform any given singular solver, despite low classification accuracy.

We believe there to be a great deal of future work in this space. Algorithm selection on SMT theories other than QF_FP could be very practical, especially for theories with high diversity core algorithms. Furthermore, on diverse datasets, considered classifiers have surprisingly low accuracy. We believe this can be improved with more refined features, but it is unclear to us what those would be. Further progress on deriving feature sets from SMT-LIB inputs could open the door for several research projects in automated reasoning and machine learning.

References

[1] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. Sunny: a lazy portfolio approach for constraint solving. *Theory and Practice of Logic Programming*, 14(4-5):509–524, 2014.

[2] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi'c, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, volume 6806 of *Lecture Notes in Computer Science*, pages 171–177. Springer, July 2011. Snowbird, Utah.

[3] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, et al. Aslib: A benchmark library for algorithm selection. *Artificial Intelligence*, 237:41–58, 2016.

[4] Christopher M Bishop. *Pattern recognition and machine learning*. springer, 2006.

[5] François Bobot-CEA, Zakaria Chihani-CEA, Mohamed Iguernlala-OCamlPro, and Bruno Marre-CEA. Fpa solver.

[6] Martin Brain, Vijay Dsilva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.

[7] Martin Brain, Florian Schanda, and Youcheng Sun. Building better bit-blasting for floating-point problems. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 79–98. Springer, 2019.

[8] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[9] Chris Cameron, Holger H Hoos, Kevin Leyton-Brown, and Frank Hutter. Oasc-2017:* zilla submission. In *Open Algorithm Selection Challenge 2017*, pages 15–18, 2017.

[10] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[11] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[12] Isaac Dooley and Laxmikant Kale. Quantifying the interference caused by subnormal floating-point values. In *Proceedings of the Workshop on Operating System Interference in High Performance Applications*, 2006.

[13] Agner Fog et al. Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus. *Copenhagen University College of Engineering*, 97:114, 2011.

[14] Ian P Gent, Christopher Jefferson, Lars Kotthoff, Ian Miguel, Neil CA Moore, Peter Nightingale, and Karen E Petrie. Learning when to use lazy learning in constraint solving. In *ECAI*, pages 873–878. Citeseer, 2010.

[15] Holger Hoos, Marius Lindauer, and Torsten Schaub. claspfolio 2: Advances in algorithm selection for answer set programming. *Theory and Practice of Logic Programming*, 14(4-5):569–585, 2014.

[16] Barry Hurley, Lars Kotthoff, Yuri Malitsky, and Barry OSullivan. Proteus: A hierarchical portfolio of solvers and transformations. In *International Conference on AI and OR Techniques in Constriant Programming for Combinatorial Optimization Problems*, pages 301–317. Springer, 2014.

[17] Yuri Malitsky. Evolving instance-specific algorithm configuration. In *Instance-Specific Algorithm Configuration*, pages 93–105. Springer, 2014.

[18] Bruno Marre, François Bobot, and Zakaria Chihani. Real behavior of floating point numbers. In *The SMT Workshop*, 2017.

[19] Mehryar Mohri, Afshin Rostamizadeh, and Ameet Talwalkar. *Foundations of machine learning*. MIT press, 2018.

[20] Jean-Michel Muller, Nicolas Brisebarre, Florent De Dinechin, Claude-Pierre Jeannerod, Vincent Lefevre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé, Serge Torres, et al. Handbook of floating-point arithmetic. 2010.

[21] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[22] J NAGELE, E Bartocci, D Beyer, PE Black, G Fedyukovich, H Garavel, A Hartmanns, M Huisman, F Kordon, M Sighireanu, et al. Toolympics 2019: An overview of competitions in formal methods. International Conference on Tools and Algorithms for the Construction and , 2019.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[24] Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In *International Conference on Principles and Practice of Constraint Programming*, pages 574–589. Springer, 2007.

[25] Stuart J Russell and Peter Norvig. *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,, 2016.

[26] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning: From theory to algorithms*. Cambridge university press, 2014.

[27] Erich Strohmaier, Jack Dongarra, Horst Simon, and Martin Meuer. June 2018.

[28] Kevin Tierney and Yuri Malitsky. An algorithm selection benchmark of the container premarshalling problem. In *International Conference on Learning and Intelligent Optimization*, pages 17–22. Springer, 2015.

[29] Mauro Vallati, Lukáš Chrpa, and Diane Kitchin. Portfolio-based planning: State of the art, common practice and open challenges. *AI Communications*, 28(4):717–733, 2015.

[30] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla-07: the design and analysis of an algorithm portfolio for sat. In *International Conference on Principles and Practice of Constraint Programming*, pages 712–727. Springer, 2007.

[31] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

[32] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2009: an automatic algorithm portfolio for sat. *SAT*, 4:53–55, 2009.

[33] Lin Xu, Frank Hutter, Jonathan Shen, Holger H Hoos, and Kevin Leyton-Brown. Satzilla2012: Improved algorithm selection based on cost-sensitive classification models. *Proceedings of SAT Challenge*, pages 57–58, 2012.