Automatic Algorithm Selection in Computational Software Using Machine Learning

Matthew C. Simpson
NC State University
Raleigh, NC 27695
Email: mcsimps2@ncsu.edu

Qing Yi
University of Colorado at Colorado Springs
Colorado Springs, CO 80918
Email: qyi@uccs.edu

Jugal Kalita
University of Colorado at Colorado Springs
Colorado Springs, CO 80918
Email: jkalita@uccs.edu

***Abstract*—Computational software programs, such as Maple and Mathematica, heavily rely on superfunctions and meta-algorithms to select the optimal algorithm for a given task. These meta-algorithms may require intensive mathematical proof to formulate, incur large computational overhead, or fail to consistently select the best algorithm. Machine learning demonstrates a promising alternative for automatic algorithm selection by easing the design process and overhead while also attaining high accuracy in selection. Two case studies are selected to demonstrate this hypothesis, namely the resultant superfunction, which computes the pairwise difference between roots of any two polynomials, and the shortest tour superfunction, which finds the least weight Hamiltonian cycle in a graph. These functions have multiple algorithms available for their computation in many mathematical software programs. Neural networks, random forests, k-nearest neighbors, and linear and RBF kernel SVMs are each trained as automatic algorithm selection tools. For the resultant superfunction, the models are trained to select algorithms based on which will give the lowest runtime for a given input. In this case, neural networks perform the best, correctly selecting the optimal algorithm out of the four available 86% of the time in Maple and 78% of the time in Mathematica. When used as a replacement for pre-existing meta-algorithms, the neural network brings about a 68% runtime improvement in Maple and a 49% improvement in Mathematica. For the shortest tour superfunction, the machine learning models are trained to minimize both runtime and approximation error subject to user specified weights on how important each aspect of performance is. Random forests outperform other machine models, attaining a 99% accuracy in selecting the optimal algorithm. When Mathematica's meta-algorithm is replaced with the random forest model, not only is the model able to select the algorithms that give the shortest tour (zero approximation error) in a given graph, but it does so while lowering the runtime by 75% compared to Mathematica's meta-algorithm.**

***Index Terms*—Resultant, Shortest Tour, Traveling Salesman Problem, Machine Learning, Meta-algorithms, Superfunctions, Algorithm Selection, AAS, Maple, Mathematica, Computational, Software**

I. Introduction

The algorithm selection problem was first formalized by Rice [1] and is stated as follows:

Given the space of all problems $\mathscr{P}$, along with an algorithm space $\mathscr{A}$ which contains all known algorithms to solve the problems in $\mathscr{P}$, determine a selection mapping $S : \mathscr{P} \to \mathscr{A}$ that maximizes performance for each problem $x \in \mathscr{P}$.

That is, if we measure performance in $\mathbb{R}^n$, where there are n dimensions of performance to take into consideration (*e.g.* runtime, memory usage, error), and if we define a performance measure $p : \mathscr{A} \times \mathscr{P} \to \mathbb{R}^n$ that maps an algorithm applied to a problem instance to its performance in $\mathbb{R}^n$, we wish to find the aforementioned selection mapping S such that for any $x \in \mathscr{P}$, it holds that $||p(S(x), x)|| \geq ||p(a, x)||$ for each $a \in \mathscr{A}$. In essence, S finds the algorithm that offers the best performance for any problem instance.

A robust solution to the algorithm selection problem is especially important for NP-Hard problems, where the algorithm runtimes can be highly variable based on the inputs to the problem. Mathematical and scientific computational software, such as Matlab, Mathematica, Maple, Sage, and NumPy, have largely resorted to employing superfunctions and meta-algorithms as a solution to this problem. Superfunctions are methods that encapsulate function calls to more specific methods to compute what the user desires. For example, the superfunction *dsolve* can be called by a Maple user to solve a system of ordinary differential equations, but it does so by, in turn, calling more specific subroutines that are available to solve such systems, such as the Taylor series method or the Rosenbrock method. To determine which specific method to call, these superfunctions use a meta-algorithm, which is responsible for making an educated choice as to which algorithm will perform the best, usually with regards to runtime.

Algorithm selection ultimately relies on properties of the inputs. Some inputs simply just won't work with certain algorithms, and other algorithms are able to provide performance enhancements and error reductions if the input is well conditioned. The latter opens up an opportunity for machine learning to be the ultimate selector of which algorithm to employ. Meta-algorithms rely on preconceived notions and rules of thumb about which algorithm should be the best in a given situation, rather than on statistical data about what has been the best approach. Designing meta-algorithms can be extremely complicated, especially to take in as many features of the inputs as possible. It is very likely some features are missed or will have to be ignored for the sake of computation. The difficulty of design is escalated when multiple aspects of the output are important, such as when both runtime and error reduction need to be taken into account instead of just one or the other. In addition, the complicated design of meta-algorithms can add large overhead to what may seem to be a

simple task. Sometimes, this overhead is more than that of the ultimate algorithm selected [2].

Machine learning can be used in practice to replace meta-algorithms by acting as a classifier to analyze important features of the input and then classify the input into which algorithm would be appropriate for it. This approach allows the program to make use of a wider feature set to make more precise decisions, compared to the smaller feature set the designers of meta-algorithms are usually forced to focus on. This added precision brings about a higher accuracy in the proportion of times the best algorithm is selected. Machine learning also avoids the need to create rules of thumb based on each feature and eases the design process needed to create an algorithm selection tool. Such an approach can be applied to any superfunction with performance dependence on inputs. For each superfunction, the important input features that affect performance need to be extracted to train the model, a general process which does not require significant tailoring to any specific problem as meta-algorithm generally do. Additionally, the problem of adding newly implemented algorithms to the pool of ones to select from becomes trivial; the machine learning model need only be retrained to account for these new algorithms, which takes a matter of seconds, compared to a total rewrite and overhauled design of meta-algorithm code. Lastly, machine learning can take into account multiple facets of runtime when choosing an algorithm, such as runtime, memory usage, and error, whereas meta-algorithms typically have to focus on only one of these aspects.

This paper aims to make headway by using machine learning as a tool for automatic algorithm selection (AAS) in computational software. Two case studies are used to providence evidence for this hypothesis: the resultant superfunction and the shortest tour superfunction. The resultant superfunction is available in most symbolic computation programs (*e.g.* Mathematica, Maple, Sage), and others with greater support for graph objects typically have a shortest tour function implemented (*e.g.* Mathematica, SAS). For these case studies, Maple and Mathematica are used to evaluate the results of machine learning and compare them to their corresponding meta-algorithm implementations. The main technical contributions of this paper are

- Developing a general approach and necessary formulations for using machine learning to automate the selection of algorithms in computational software
- Demonstrating, through the resultant and shortest tour superfunctions, the success of machine learning as a tool for AAS
- Empirically comparing the performance of machine learning models to that of Maple's and Mathematica's meta-algorithms

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 introduces the mathematical background of the resultant and the algorithms used for its computation. Maple's meta-algorithm for choosing amongst these algorithms is also discussed, as well as the features and datasets used for training multiple machine learning models. Section 4 does the same for the shortest tour superfunction. Section 5 evaluates the accuracy of different machine learning approaches and compares these results to those of Maple and Mathematica, the main ideas of which are again summarized in Section 6.

II. Related Work

Meta-algorithms are not the only approach to the algorithm selection problem. The algorithm portfolio paradigm [3] was an early attempt at a solution, which, in effect, selects a portion of all available algorithms and runs them in parallel until one of them finishes. The parallel computation causes a large resulting overhead, but nevertheless, this method shines for certain problem classes with heavy-tailed runtime distributions. In these cases, the algorithm portfolio paradigm is more advantageous than running a single algorithm [4]. Dynamic algorithm portfolios [5] provide a step forward from traditional algorithm portfolios by running a set of algorithm in parallel, but then iteratively updating the priority of each process by how well each is performing. The biggest problem here, however, is developing and implementing a measure in each algorithm that allows one to judge the current progress made. More importantly, this measure needs to designed such that it allows for a fair comparison between different algorithms.

For many end users in scientific and mathematical software, these parallel computing approaches are not feasible given their large overhead. Although these approaches minimize runtime, no regard is given to the resource management aspect of performance. Because of this, meta-algorithms have been selected as the preferred tool in computational software.

The idea of using statistical and machine learning techniques for the selection of algorithms is not unheard of. Brewer [6][7] brought about the idea of using regression for performance predictions by using linear fitting to predict the runtime of different implementations of multiprocessor libraries on unseen architectures. In the field of meta-learning, Bradzil et al. [8] applied this technique using an Instance-Based Learning approach to select appropriate learning algorithms for different sets of problems. Similarly, the study in this paper continues to build upon this idea in the context of computational software.

III. The Resultant Case Study

To demonstrate the overarching hypothesis that machine learning can be a better alternative to meta-algorithms for use in computational software, a popular superfunction was chosen as the first case study, namely the resultant. The resultant is a fundamental tool used in computer algebra, algebraic geometry, algebraic cryptography, and elimination theory, and it finds higher level use in algorithms for integration, solving systems of nonlinear equations, computing the discriminant of two polynomials, analyzing greatest common divisors, and so forth.

A. The Resultant

Given two polynomials $a, b \in \mathbb{F}[x_1, x_2, \ldots, x_k]$, where $\mathbb{F}$ denotes an integral domain, of degree n and m respectively, they can be written with respect to the variable x_l, $1 \leq l \leq k$, as $a(x_l) = a_n x_l^n + \ldots + a_1 x + a_0$ and $b(x_l) = b_m x_l^m + \ldots + b_1 x_l + b_0$, where $a_i, b_i \in \mathbb{F}[x_1, \ldots, x_{l-1}, x_{l+1}, \ldots, x_k]$. The resultant function taken with respect to x_l is defined as the determinant of Sylvester's matrix [9][10]:

$$S_{x_l}(a,b) = \begin{bmatrix} a_n & a_{n-1} & \ldots & a_0 & 0 & \ldots & 0 \\ 0 & a_n & \ldots & a_1 & a_0 & 0 & \ldots \\ \vdots & \ldots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & a_n & a_{n-1} & \ldots & a_0 \\ b_m & b_{m-1} & \ldots & b_0 & 0 & \ldots & 0 \\ 0 & b_m & \ldots & b_1 & b_0 & 0 & \ldots \\ \vdots & \ldots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \ldots & 0 & b_m & b_{m-1} & \ldots & b_0 \end{bmatrix}$$

The resultant turns out to be of interest because computing the determinant of the Sylvester's matrix is equivalent to the following :

$$Res_{x_l}(a,b) = det(S_{x_l}(a,b)) = a_n^m b_m^n \prod_{\substack{(r_a, r_b): \\ a(r_a)=b(r_b)=0}} (r_a - r_b)$$

For monic polynomials, this is the product of the pairwise difference in the roots $r_a, r_b \in \mathbb{F}[x_1, \ldots, x_{l-1}, x_{l+1}, \ldots, x_k]$ of each polynomial. This becomes especially important for handling the roots of polynomials in an algebraic fashion, that is, without actually having to explicitly compute their value. Thus, in computation, the roots are never solved for, because that is a problem in and of itself and defeats the purpose of handling the roots algebraically.

Although more exist, four algorithms are generally used to compute the resultant: Sylvester's matrix, Bezout's matrix [11], Collin's modular resultant [12], and Brown's subresultant pseudo-remainder sequences [13]. All four of these algorithms are available in Mathematica and Maple.

The computation of the resultant via Bezout's matrix is quite similar to doing so with Sylvester's matrix, given above. Bezout's matrix is defined as $B(a,b) = (b_{ij})_{i,j=1,\ldots,\max[n,m]}$ with elements $b_{ij} = \sum_{k=1}^{\min[i,n+1-j]} a_{j+k-1}b_{i-k} - a_{i-k}b_{j+k-1}$. The determinant of this matrix is a multiple of the resultant of a and b.

Collin's modular resultant algorithm derives from the idea that taking the determinant of a matrix, such as Sylvester's matrix, only requires addition and multiplication operations. Thus, it makes sense to take advantage a homomorphism $\phi : X \to Y$, where it holds $\phi(x_1 + x_2) = \phi(x_1) + \phi(x_2)$ and $\phi(x_1 x_2) = \phi(x_1)\phi(x_2)$ for any $x_1, x_2 \in X$. In this case, it also holds then for a matrix $Q = (q_{ij})$ with elements $q_{ij} \in X$ that $\phi(det(Q)) = det(\phi(Q))$. Collin originally applied this result to the domain of integers $\mathbb{Z}$ using the mapping $\phi : \mathbb{Z} \to \mathbb{Z}_p$, where p is a prime integer, to take advantage of the fact that if $|det(Q)| < \frac{p}{2}$, then $\phi(det(Q)) = det(Q)$, and thus, by the previous homomorphic equality, $det(\phi(Q)) = det(Q)$. Typically, to make use of smaller primes, several primes and homomorphic mappings are used to do the reduced calculations, the results of which can then be recombined using the Chinese remainder theorem.

Brown's subresultant algorithm is a generalization of the Euclidean algorithm [14] for computing the greatest common divisor of two integers or polynomials. For polynomials $a, b \in \mathbb{F}[x_1, x_2, \ldots, x_k]$, assume, without loss of generality, that $deg(a) \geq deg(b)$. When the division algorithm holds, a can be written as $a = b \cdot q + r$, where $q, r \in \mathbb{F}[x_1, x_2, \ldots, x_k]$ and are called the quotient and remainder, respectively. If the division algorithm does not hold, pseudo-remainders are used instead by introducing a constant multiplier to a. The Euclidean algorithm makes use of the fact that $\gcd(a,b) = \gcd(b,r)$, so we can recursively apply this reduction until the remainder is 0. The subresultant algorithm uses a similar equality but applied to subresultants, which come from submatrices of Sylvester's matrix. These are denoted $Res_{x_l}^j(a,b)$ as the j-th order subresultant, which effectively eliminates j rows and columns from Sylvester's matrix. The following equality then holds:

$$Res_{x_l}^j(a,b) = (-1)^{(n-j)(m-j)} b_m^{n-deg(r)} Res_{x_l}^j(b,r)$$

This allows us to iteratively simply the problem, just like the Euclidean algorithm.

B. Meta-algorithms

Mathematica's meta-algorithm to select among the four available algorithms is hidden, but Maple's is available in the documentation on the resultant superfunction and by using the *showstat* command to view the source code. For univariate and bivariate polynomials with rational coefficients (including integral coefficients, even though the two are stored in memory differently), Maple uses modular methods for high degree polynomials, whereas the subresultant algorithm is used for those with lower degrees. In all other cases, Bezout's matrix is used. It's worth noting that, even though Maple offers Sylvester's matrix as an option, it is never considered by the meta-algorithm. Without a doubt, this is because, in most cases, taking the determinant of a smaller matrix, such as Bezout's matrix, is faster than taking the determinant of a larger matrix, such as Sylvester's matrix. However, this assumption ignores important factors, such as the computation of the elements in Bezout's matrix, whether any elements are zero, whether floating point numbers are being used, the kernel of the program, and so on. These factors can regularly make Sylvester's matrix a more viable method; in fact, for randomly sampled polynomials, the proportion of times Sylvester's matrix outperforms all other algorithms is close to the same proportion of times Bezout's matrix does so too.

C. Machine Learning Formulations

Machine learning can be applied to the algorithm selection problem by using classification to categorize a pair of input polynomials into which algorithm will work for best for them. This classification relies only on a set of attributes or features that are taken from a quick analysis of the two polynomials a and b passed in as arguments during the function call $resultant(a, b, x_l)$, the resultant of a and b with respect to the variable x_l. Feature engineering can be completed by analyzing the source code for implemented algorithms or by having a general understanding of each algorithm and then determining the fundamental features of the inputs on which the runtime complexity depends.

The basic features on which to classify are generally easy to spot. For example, it is clear to see that all four available algorithms to compute the resultant depend on the degrees of the input polynomials, although in different manners; the dimensions of Bezout's and Sylvester's matrices directly depend on the degrees of the polynomials, and the coefficient bounds of Collin's modular method changes with the degrees of the polynomials, as well as the number of iterations that occur in the subresultant algorithm.

It is usually the case that algorithms scale differently with regards to changes in any given feature. For example, the cost to take the determinant of Sylvester's matrix scales polynomially with the degrees of the inputs, whereas the coefficient bound in the modular resultant algorithm scales factorially. It's also possible that an algorithm's runtime may change with a given feature, whereas another algorithm's may not change at all, as might happen with the fact that Sylvester's matrix could care less about the polynomial ring, whereas the modular method's runtime may change significantly.

After enumerating many of these features, the list was narrowed down using the ReliefF algorithm [15] to 18 attributes from an initial size of 30. The ReliefF algorithm iteratively takes a random feature vector from the training data and creates a weight for each feature by analyzing the k nearest-hits, which are the closest feature vectors under the L1 norm with the same classification, and the k nearest-misses from each different class, which are the closest feature vectors with a different classification. The full list of attributes used for the resultant case study, along with their descriptions, is given in Table I.

As for labeling, there were four target classes for each of the four available algorithms. Classification into one of these categories means the resultant algorithm corresponding to that class gives the best performance in terms of runtime.

The dataset consisted of 18,346 randomly generated polynomials, which were randomly paired into 9,173 inputs, as the resultant function takes two polynomials as a single input. Since the learning was supervised, the output class for each input in the dataset was obtained by running each of the four algorithms, which are already implemented in Maple and Mathematica, and selecting the resultant algorithm that gave the least runtime over an average of thirty runs.

Neural networks, random forests, k-nearest neighbors, and SVMs with linear and RBF kernels were trained on the given data. The neural network was built with Matlab's pattern recognition tool and consisted of a single hidden layer with 10 sigmoid hidden neurons and a softmax output layer. These formed a feed-forward network that was trained with scaled conjugate gradient backpropogation. The data for the neural network was split into 70% training, 15% validation, and 15% testing.

The random forest model was built based off the model in [16] with 50 random trees that, at each node, split on a random selection of five features. Forests with more trees did not offer significant gains in accuracy compared to the associated cost in runtime, and splitting among less than five features caused a loss in accuracy. For the k-nearest neighbors model, classification was done based solely on the nearest neighbor, since using multiple neighbors caused the accuracy to drop off steeply. A linear search with the Euclidean distance norm was used to find the nearest neighbor. Both the random forest and the k-nearest neighbors model were trained with 10-fold cross validation. Lastly, the SVM model was trained with both a linear and a RBF kernel through libsvm's interface. For these three methods, data was divided into 80% training and 20% testing.

IV. The Shortest Tour Case Study

The second case study chosen to evaluate machine learning's capabilities as an automatic algorithm selection tool was the shortest tour superfunction, which chooses amongst several algorithms to solve the traveling salesman problem.

A. The Traveling Salesman Problem

Given an undirected, weighted graph G, the traveling salesman problem (TSP) asks for the route with the least weight that visits each vertex once and returns to the starting vertex. G is often taken to be a simple graph, since any parallel edges can be eliminated to the one with the least weight and loops are never used in the final tour. Often, the problem may take a set of points as an input, with the goal now attempting to find the shortest route that visits each point given that the distance between any two points is determined by some metric, such as the Euclidean norm or the Manhattan distance. This is just the original problem applied to a complete graph, where any two points are connected with edge weights determined by their physical distance from each other.

Mathematica provides implementations of the following algorithms to solve the traveling salesman problem for graph objects, which are available through its *FindShortestTour* superfunction:

- Greedy [17]
- Greedy Cycle [17]
- Integer Linear Programming (ILP) [18]
- Simulated Annealing [19]
- Or-Opt [20]
- Two-Opt [20]

TABLE I
POLYNOMIAL FEATURES

No.	Feature	Description
1.	True/False: $a, b \in \mathbb{Z}[x_1, \ldots, x_k]$	Whether or not polynomials have all integer coefficients
2.	True/False: $a, b \in \mathbb{Q}[x_1, \ldots, x_k]$	Whether or not polynomials have all rational coefficients
3.	True/False: a or b has floating point coefficients	Whether or not computations will require floating point precision
4.	Number of indeterminants	Number of variables in both polynomials combined
5.	$deg(a)$	Degree of a with respect to x_l
6.	$deg(b)$	Degree of b with respect to x_l
7.	Number of terms in a	Self explanatory
8.	Number of terms in b	Self explanatory
9.	Sparsity rating of a	(Number of nonzero coefficients of a)/$(deg(a) + 1)$
10.	Sparsity rating of b	(Number of nonzero coefficients of b)/$(deg(b) + 1)$
11.	Number of algebraic coefficients in a	The number of terms that consist of a variable besides x_l
12.	Number of algebraic coefficients in b	The number of terms that consist of a variable besides x_l
13.	Proportion of algebraic coefficients in a	What portion of coefficients are purely algebraic in a
14.	Proportion of algebraic coefficients in b	What portion of coefficeints are purely algebraic in b
15.	$lcoeff(a)$	The numeric coefficient of the highest order term in a
16.	$lcoeff(b)$	The numeric coefficient of the highest order term in b
17.	Numeric coefficient with smallest magnitude out of a and b	Self explanatory
18.	Numeric coefficient with largest magnitude out of a and b	Self explanatory

More algorithms are available that are specialized for sets of points instead of graph objects. However, the results in this paper only focus on these six algorithms; that is, they only focus on graph inputs. The same machine learning approach can be applied to classify sets of points into which algorithm is most appropriate for them.

Mathematica's documentation does not make clear which greedy heuristics it employs in its greedy and greedy cycle algorithms. However, it is likely the greedy approach is simplest the nearest neighbors approach that, at each vertex, selects the nearest neighbor as the next point in the tour to proceed to. The greedy cycle algorithm is likely an insertion algorithm that starts with tour consisting of a randomly selected vertex and its nearest neighbor and then iteratively inserting a new vertex k into the tour between two adjacent nodes i, j that already exist in the tour such that the cost of the sum of edges $i\bar{k}$ and $\bar{k}j$ are minimized.

ILP formulates the TSP problem as a minimization problem. [?] describes the problem as follows: For each edge $e \in E(G)$, we define x_e as taking the value 1 if e is part of the tour and 0 otherwise. In addition, $c_e \in \mathbb{R}$ defines the cost of e. Letting $\delta(S) = \{\bar{ij} \in E(G) \mid i \in S, j \notin S\}$ for any set $S \subset E(G)$, we have the following optimization to perform:

$$\begin{aligned} \text{Minimize} \quad & \sum_{e \in E(G)} c_e x_e \\ \text{s.t.} \quad & \sum_{e \in \delta(v)} x_e = 2 \ \forall v \in V \\ & \sum_{e \in \delta(S)} x_e \geq 2 \ \forall S \subset V, 2 \leq |S| \leq |V| - 1 \end{aligned}$$

Simulated annealing also treats the TSP problem as a minimization problem. It begins with a random tour through all vertices and then moves on to selecting a new tour from the neighbors of the existing tour. If the new tour is better than the old tour, it is chosen as the preferred tour and the process continues. If it is not better, then it is accepted as the new tour with a probability dependent on both the difference between tour lengths of the new and old tours as well as the temperature or energy of the annealing process; higher temperatures correspond to a higher probability of acceptance. This temperature is then lowered a bit and the process repeated until a minimum is reached. This process can often result in local minimums rather than global minimums.

Or-opt and Two-opt are both heuristics that begin with an attempted solution, a Hamiltonian cycle that may or may not be the least cost tour. Or-opt is a chain exchange method that repeatedly moves chains of three consecutive vertices to different locations until such movements can no longer provide cost improvements. This then continues with two vertices and then a single vertex. Two-opt is an edge exchange method that iteratively removes two edges from the initial tour and then attempts to reconnect the two remaining chains in a way that lowers the total cost from the previous tour.

B. Meta-algorithm

Like with the resultant superfunction, Mathematica's meta-algorithm implementation is not described in the documentation and is not available as source code. However, it is known that the meta-algorithm guarantees the shortest tour, not an approximation of the shortest tour. However, it is not likely that Mathematica simply defaults to using integer linear programming. The disparity between the time it takes Mathematica's meta-algorithm to compute the shortest tour and the time it takes a direct call to integer linear programming varies too widely for such a statement to be true. What is more likely is that Mathematica makes use of running several algorithms as an attempt to find a solution or opts to run an approximation algorithm in a situation where it is guaranteed it will find the shortest tour.

C. Machine Learning Formulations

As previously mentioned, the greedy, greedy cycle, Or-Opt, Two-Opt, and simulated annealing algorithms all approximate solutions that come with the benefit of a faster runtime than

integer linear programming, which gives an exact solution. However, it is possible for some of the methods to fail to find any solution to the problem at all while the others succeed. Mathematica immediately makes this known to the user.

Since both approximation and exact algorithms are available to choose from in Mathematica, it would not make sense to consider only runtime or approximation error as the sole performance measure on which to base the choice in algorithm. If runtime was the only consideration, the nearest neighbors approach would be the choice every time, just with the cost of large approximation error. Likewise, if approximation error was the only factor of importance, then ILP would be the best choice with zero approximation error but a hefty runtime, assuming it is able to generate a solution. Therefore, it is important to consider both of these factors when deciding what algorithm should be deemed the best algorithm.

To solve this issue, a user specified parameter was used that would allow a user to define how important runtime and approximation error should weigh against each other when the shortest tour superfunction was called. This weight factor, denoted γ, holds a value between 0 and 1, 0 meaning all consideration should be given to minimizing runtime and 1 meaning all consideration should be given to minimizing approximation error. A value of 0.5 would mean both factors should be equally weighted against each other. Machine learning can use this parameter to select an algorithm based on how important runtime and approximation error are to the user, a naive approach to multi-objective classification.

2332 randomly generated graphs, consisting of up to 50 vertices, edge weights between 1 to 100, and all guaranteed of having a Hamiltonian cycle, were used as a training set. Each of the six algorithms were run on each of the 2332 inputs, and their runtimes and approximation errors were recorded. For each algorithm, the percent difference from the best runtime out of all algorithms was noted, as well as the percent difference from the best approximation error out of all algorithms. Then for any algorithm that produced a solution, it's performance would be defined by $\gamma \times$(% Diff. Approx. Err.) + (1-γ) $\times$ (% Diff. Runtime). The optimal algorithm for a given weight factor is the one that has the smallest value of this measure.

For different values of γ, the optimal algorithm often changes. To train machine learning to learn this pattern, the 2332 randomly generated graphs were each analyzed with values of γ in the set $\{0, 0.1, 0.2, \ldots, 1\}$. In this sense, γ forms a feature of the input that is used for classification because the same graph with different values of γ leads to different classifications. This process ultimately creates a dataset of 25,652 examples on which to train and test.

In addition to the weight factor γ, the features used in classification are described in Table II. In total, there were 21 features to describe a given input. It is highly likely some of the features are repetitive (have information that are given by other features) or are not necessary, as this list of features has yet to be narrowed down by a feature selection process.

Like the resultant superfunction, neural networks, random forests, k-nearest neighbors, and linear and RBF SVMs were all trained on the given data. All setups for these machine learning models were exactly the same as for the resultant case study, except the neural network had 50 hidden neurons instead of 10 for increased performance.

V. Experimental Results

To train and test the machine learning models, all computation was done on a build with an Xeon E5-2420 CPU, Matrox G200eR2 video controller, and 16 GB of RAM, running 64 bit CentOS 6.8 with an installation of Maple 2015.1 and Mathematica 10.0.2.

A. The Resultant

1) Setup: The performance of machine learning as an automatic algorithm selection tool for the resultant superfunction was compared to Maple's and Mathematica's meta-algorithms. More tools for comparison exist, such as Sage, but these tend to default to simply computing the determinant of Sylvester's matrix instead of choosing amongst the different algorithms available. To test the benefits of machine learning, neural networks, random forests, k-nearest neighbors, and SVMs were each trained, and the two most accurate of which, namely neural networks and random forests, were used to replace Maple's and Mathematica's meta-algorithm to select the ultimate resultant algorithm used. These two were then run and timed over several thousand problem inputs. Note, however, that the output class for a pair of inputs was obtained by running all algorithms implemented by the given program and taking the one that gave the least runtime. Since Maple and Mathematica have different kernels which are optimized for different operations, the best algorithm when run under Maple may not be the same as the best algorithm when run under Mathematica for a given input. Thus, it would not be wise to use the machine learning model trained on the best algorithms determined by Maple's runtimes as a replacement for Mathematica's meta-algorithm. Keeping with this paradigm, two datasets were generated, one of which based its output classes on the best algorithm when run under Maple and the other when run under Mathematica.

2) Accuracy: The accuracies, or proportions of the time when the algorithm with the least runtime out of all four available resultant algorithms was correctly chosen, for each machine learning approach used are given in Table III. For training on the data generated under the best algorithm choices determined by runtimes in Maple, neural networks outperformed the other machine learning models, correctly selecting the best algorithm 86.63% of the time on the testing data (86.17% and 85.54% for training and validation respectively). Random forests managed to attain 80.71% accuracy, but k-nearest neighbors, RBF kernel SVMs, and linear kernel SVMs all lagged behind at 77.46%, 76.68%, and 76.63% accuracy, respectively.

However, these accuracies indicate that the associated machine learning models are vast improvements over the use of meta-algorithms. Maple's meta-algorithm selects the best

TABLE II
GRAPH FEATURES

No.	Feature	Description
1.	γ	Weight factor - how important runtime or approximation error is for selecting an algorithm
2.	$\lvert V(G)\rvert$	Number of vertices
3.	$\lvert E(G)\rvert$	Number of edges
4.	$Max[cost(e) \mid e \in E(G)]$	Maximum edge weight in G
5.	$Min[cost(e) \mid e \in E(G)]$	Minimum edge weight in G
6.	$\lvert E(G)\rvert / \binom{\lvert V(G)\rvert}{2}$	The proportion of edges used compared to the complete graph on $\lvert V(G)\rvert$ vertices
7.	$\sum_{e \in E(G)} cost(e)$	Sum of weights in G
8.	$Average[cost(e) \mid e \in E(G)]$	Average edge weight
9.	$StandardDeviation[cost(e) \mid e \in E(G)]$	Standard Deviation of edge weights
10.	$Median[cost(e) \mid e \in E(G)]$	Median edge weight
11.	$\left(\sum_{e \in E(G)} cost(e)\right) / \left(100 \cdot \binom{\lvert V(G)\rvert}{2}\right)$	Density of edges, where 100 is the maximum possible edge weight
12.	$Max[deg(v) \mid v \in V(G)]$	Maximum degree in G
13.	$Min[deg(v) \mid v \in V(G)]$	Minimum degree in G
14.	$Average[deg(v) \mid v \in V(G)]$	Average degree of a vertex
15.	$StandardDeviation[deg(v) \mid v \in V(G)]$	Standard deviation of the degrees of vertices in G
16.	$Median[deg(v) \mid v \in V(G)]$	Median vertex degree
17.	$Max[\sum_{e \in N(v)} cost(e) \mid v \in V(G)]$	Maximum sum of costs of incident edges for any vertex
18.	$Min[\sum_{e \in N(v)} cost(e) \mid v \in V(G)]$	Minimum sum of costs of incident edges for any vertex
19.	$Average[\sum_{e \in N(v)} cost(e) \mid v \in V(G)]$	Average sum of costs of incident edges for any vertex
20.	$StandardDeviation[\sum_{e \in N(v)} cost(e) \mid v \in V(G)]$	Standard deviation of costs of incident edges for any vertex
21.	$Median[\sum_{e \in N(v)} cost(e) \mid v \in V(G)]$	Median of costs of incident edges for any vertex

TABLE III
ACCURACIES FOR VARIOUS MACHINE LEARNING MODELS FOR THE RESULTANT

Model	Accuracy on Maple data	Accuracy on Mathematica data
Neural Networks	86.63%	78.24%
Random Forests	80.71%	75.97%
KNN	77.46%	69.03%
RBF SVM	76.68%	70%
Linear SVM	76.63%	67%

algorithm out of all four available only 58% of the time. (The accuracy of the meta-algorithm can be determined by looking at the source code or the *userinfo* output to see when a specific algorithm has been called.) However, as previously discussed, considering the meta-algorithm only ever opts for three out of the four available (namely, it ignores Sylvester's matrix), the accuracy in choosing the best out of three is 72%. Unfortunately, in close to 15% of cases, Sylvester's matrix proves to be the best choice, so although Maple can select amongst the modular, subresultant, and Bezout algorithms with a 72% accuracy, this choice is still not the best 15% of the time. Nevertheless, in either accuracy measurement, all tested machine learning models outperformed Maple's meta-algorithm.

Machine learning's performance dropped significantly when tested against the data generated under Mathematica. Neural networks only attained a 78.24% accuracy, and random forests came close with a 75.97% accuracy. K-nearest neighbors, RBF SVMs, and linear SVMs all underperformed at 69.03%, 70%, and 67% accuracy respectively. Unfortunately, Mathematica hides the implementation of its meta-algorithm and provides no way to see which algorithm was ultimately selected by its meta-algorithm, so the accuracy of Mathematica's choices remain unknown.

3) Time Speedup: Although there are obvious differences in accuracies between machine learning models and meta-algorithms, these differences are compounded by significantly faster runtimes when machine learning models are used to replace meta-algorithms during the selection of algorithms stage. When applied to a random sample of several thousand inputs, Maple was able to compute the resultant of all inputs in 37,783 seconds with its original meta-algorithm, whereas using the neural network as the selection tool when the resultant superfunction was called yielded a total runtime of only 12,097 seconds, a 68% decrease in runtime. Similarly, in Mathematica, the neural network brought about a 49% decrease in runtime. Random forests, on the other hand, only brought a 46% decrease in runtime to the same sample in Maple and a 37% decrease in Mathematica. Since neural networks have such a better runtime improvement compared to random forests, even though their accuracies only differ by close to 6%, it appears as though when the neural network made an incorrect algorithm choice, the decision it did make was not as bad as when the same situation occurred for random forests. In the case of random forests, the incorrectly chosen algorithm was typically also not the second best algorithm choice. Nevertheless, the high accuracies still resulted in significant performance gains. Since running these tests takes large amounts of server time and resources, the runtime results are limited to just neural networks and random forests and are summarized in Table IV.

It serves to note that there is evidence that Mathematica's meta-algorithm uses pre-processing to speed up the computation of the ultimately selected algorithms, whereas a direct call to a specific resultant algorithm does not use this pre-processing. This claim comes from the fact that, in occasional instances in the dataset, a call to Mathematica's

TABLE IV
RUNTIME IMPROVEMENTS USING MACHINE LEARNING IN PLACE OF META-ALGORITHMS FOR THE RESULTANT

Model	Runtime Improvement in Maple	Runtime Improvement in Mathematica
Neural Networks	68%	49%
Random Forests	46%	37%

meta-algorithm yields a faster runtime than single-handedly calling any one of the four algorithms directly. Yet, despite this disadvantage, using machine learning as an automatic algorithm selection tool still manages to outperform Mathematica's meta-algorithm by a significant factor.

B. Shortest Tour

1) Setup: Mathematica was used as a tool for comparison against the machine learning models trained on the TSP problem. For each input and each value of $\gamma \in \{0, 0.1, 0.2, \ldots, 1\}$, the timings of each of the six algorithms available to compute the shortest tour of a graph were recorded as well as the length of the tour the generated. Each algorithm's performance was gauged with the previously discussed performance measure $\gamma \times$ (% Diff. Approx. Err.) + (1-γ) $\times$ (% Diff. Runtime), where the percent differences are measured with respect to the best approximation errors and runtimes provided out of all the algorithms. The algorithm with the least value given by this measure was deemed the optimal algorithm for the given graph input and value of γ. This process resulted in 25,652 examples to train and test against.

For the value of $\gamma = 1$, there were often multiple algorithms that have the same performance measure. That is, in this particular case, they all returned the shortest possible tour, as runtime is not considered when $\gamma = 1$. When this happened, ties were broken according in the following order, from the highest preferences to lowest: Greedy, Greedy Cycle, Two-Opt, Integer Linear Programming, Or-Opt, Simulated Annealing. The reason for this is that the former algorithms give better runtimes compared to the latter algorithms.

2) Accuracy: The proportions of times different machine learning models were able to select the optimal algorithm based on the features of the input graph and the value of $\gamma$are given Table V. All machine learning models except for linear SVMs demonstrated exceedingly high accuracies. Random forests and k-nearest neighbors reached 99.6% and 99.62% accuracies, and neural networks and RBF SVMs lagged slightly behind at 96.81% and 95.61% accuracies, respectively. A linear kernel for SVMs, on the other hand, did not sufficiently separate the data and resulted in an accuracy of 38.3%.

3) Time Speedups: To test how well machine learning fares at automatic algorithm selection, the total runtimes and path lengths were analyzed from a random sample of 1500 graphs using the algorithm choices decided upon by neural networks, random forests, and Mathematica's meta-algorithm. These runtimes are given in Table VI. For neural networks and random forests, the percentage by which they improve upon the meta-algorithm's runtime is also given. The sum of the all the tour lengths generated by the choice of algorithms from each tool is recorded in Table VII. Note that shortest possible sum of tour lengths is 73,076, which Mathematica attains in all possible cases at the cost of a higher runtime. Since Mathematica's meta-algorithm has no dependence on γ, the runtimes and sum of tour lengths remain constant for each value of γ.

TABLE V
ACCURACIES FOR VARIOUS MACHINE LEARNING MODELS FOR THE SHORTEST TOUR

Model	Accuracy
Neural Networks	96.81%
Random Forests	99.6%
KNN	99.62%
RBF SVM	95.61%
Linear SVM	38.3%

As expected, as the value of γ increases, the runtimes using machine learning increase while the approximation error decreases. A clear benefit from using a weight factor is that it allows the user to control which is more important in the computation: the runtime, the approximation error, or a combination of both. However, the benefits of using machine learning as a tool for AAS is best demonstrated in the case where $\gamma = 1$. In this case, no explicit attention is given to runtime; rather, the machine learning models have been trained to solely focus specifically on minimizing approximation error. As seen in Table VII, random forests attain the shortest possible tour length in every case when $\gamma = 1$, corresponding to a zero approximation error. This matches Mathematica's meta-algorithm's approximation error; however, because of the way machine learning breaks ties between candidate algorithms, random forests also offer a 75.02% improvement in runtime, as seen in Table VI. Not only is the random forest getting the exact solution like Mathematica's meta-algorithm does, but it is doing so significantly faster. Whereas Mathematica can only manage to focus on one aspect of performance, machine learning is able to balance both runtime and approximation error and see performance improvements at the same time.

VI. CONCLUSION

The results in this paper support the hypothesis that machine learning provides a better alternative for automatic algorithm selection in computational software. When used as a replacement for the resultant meta-algorithms in Maple and Mathematica, neural networks bring about 68% and 49% runtime improvements, respectively. When used in the context of the traveling salesman problem, random forests can weigh multiple facets of performance, specifically runtime and approximation error, against each other using a user specified parameter in order to choose an appropriate algorithm. Machine learning thus allows algorithm selection tools to take into account multiple objectives, whereas meta-algorithms can typically only focus on one performance aspect. However, even when minimizing approximation error in the traveling

TABLE VI
RUNTIMES FOR DIFFERENT VALUES OF γ

γ	**Runtime Under Mathematica (s)**	**Runtime Under Neural Networks (s)**	**Percent Improvement Under Neural Networks**	**Runtime Under Random Forests (s)**	**Percent Improvement Under Random Forests**
0	1797.67	47.989	97.33%	47.871	97.34%
0.1	1797.67	48.080	97.33%	47.871	97.34%
0.2	1797.67	48.025	97.33%	47.876	97.34%
0.3	1797.67	48.026	97.33%	48.655	97.29%
0.4	1797.67	48.822	97.28%	47.893	97.34%
0.5	1797.67	124.897	93.05%	125.579	93.01%
0.6	1797.67	372.427	79.28%	373.926	79.2%
0.7	1797.67	400.618	77.7%	400.429	77.73%
0.8	1797.67	410.127	77.19%	603.909	66.41%
0.9	1797.67	449.523	75%	448.865	75.03%
1	1797.67	449.739	74.98%	448.991	75.02%

TABLE VII
SUM OF TOUR LENGTHS FOR DIFFERENT VALUES OF γ

γ	**Sum of Tour Lengths Under Mathematica (s)**	**Sum of Tour Lengths Neural Networks (s)**	**Percent Error Under Neural Networks**	**Sum of Tour Lengths Random Forests (s)**	**Percent Error Under Random Forests**
0	73076	164127	124.6%	164127	124.6%
0.1	73076	164127	124.6%	164127	124.6%
0.2	73076	164127	124.6%	164351	124.9%
0.3	73076	164127	124.6%	164299	124.83%
0.4	73076	164299	124.83%	163998	124.42%
0.5	73076	150199	105.43%	149776	104.96%
0.6	73076	92635	26.77%	92855	27.07%
0.7	73076	82738	13.22%	82738	13.22%
0.8	73076	81407	11.40%	97766	33.8%
0.9	73076	73439	0.5%	73084	0.01%
1	73076	73451	0.51%	73076	0%

salesman problem is the only focus, machine learning can still output the shortest possible tour (zero approximation error) just like Mathematica's meta-algorithm, but with a 75% decrease in runtime as a bonus side-effect. Even larger decreases in runtime are observed when larger approximation errors are allowed by the user.

The overall methodology developed in this paper emphasizes the fact that the process of using machine learning as a tool for automatic algorithm selection is not only straightforward but highly effective. As an example, since most resultant applications require the computation of the resultant dozens or hundreds of times, the runtime improvement would be quite noticeable. In fact, even for a single pair of input polynomials of which to compute the resultant, it is not uncommon for the possible runtimes to range from several seconds to a couple minutes depending on which algorithm is selected. The same situations appear for traveling salesman problems. The high accuracy achieved by machine learning avoids such excessive runtimes.

What remains in the works is to expand this approach to more case studies and explore better formulated multi-objective machine learning models that would allow the optimization of different facets of performance. This would enable an algorithm selection model that can more effectively select the best algorithm based on not only runtime performance, but also based on memory constraints, least error, and so on. For instance, evolutionary multi-objective optimization algorithms, such as NSGA-II [21] and SPEA-II [22], may be easily used to perform such computation, although the time required to do so may not be acceptable but needs to be empirically determined. All in all, the results in this paper have built a foothold for further exploration into the benefits of machine learning as a tool for automatic algorithm selection, especially in computational software.

ACKNOWLEDGEMENT

This work was funded under NSF grant 1359275. The authors would also like to thank Nathan Harmon for his valued input on the project.

REFERENCES

[1] J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15, 1976.
[2] Wolfram algorithmbase. https://www.wolfram.com/algorithmbase/.
[3] R. M. Lukose B. A. Huberman and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, 1997.
[4] C. P. Gomes and B. Selman. Heavy-tailed phenomena in satisfiability and contstraint satisfaction problems. *J. Autom Reason*, 24(1-2):67–100, 2000.
[5] M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47:295–328, 2006.

[6] E. A. Brewer. *Portable high-performance supercomputing: high-level platform-dependent optimization*. PhD thesis, Massachusetts Institute of Technology, 1994.

[7] E.A. Brewer. High-level optimization via automated statistical modeling. *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 80–91, 1995.

[8] C. Soares P. Bradzil and D.C.J. Pinto. Ranking learning algorithms: Using ibl and meta-learning on accuracy and time results. *Machine Learning*, 50(3):251–277, 2003.

[9] S.R. Czapor K.O. Geddes and G. Labahn. *Algorithms for Computer Algebra*. Kluwer Academic Publishers, 1992.

[10] A. G. Akritas. Sylvester's forgotten form of the resultant. *Fibonacci Quarterly*, pages 325–332, 1993.

[11] R. N. Goldman E. Chionh, M. Zhang. Fast computation of the bezout and dixon resultant matrices. *Journal of Symbolic Computation*, 33:13–29, 2002.

[12] G. E. Collins. The calculation of multivariate polynomial resultants. *SYMSAC*, pages 212–222, 1971.

[13] W. S. Brown. The subresultant prs algorithm. *ACM Transactions on Mathematical Software*, 4(3):237–249, 1978.

[14] W. S. Brown. On euclid's algorithm and the computation of polynomial greatest common divisors. *Journal of the Association for Computing Machinery*, 18(4):478–504, 1971.

[15] M. Robnik-Sikonja I. Kononenko and U. Pompe. Relieff for estimation and discretization of attributes in classification, regression, and ilp problems. 1996.

[16] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[17] S. Goyal. A survey on traveling salesman problem.

[18] Sas/or(r) 9.22 user's guide. http://support.sas.com/documentation/cdl/en/ormpug/63352/HTML/default/viewer.htm#ormpug_milpsolver_sect020.htm.

[19] D. Bookstaber. Simulated annealing for traveling salesman problem. *SAREPORT.nb*.

[20] S. Deneault G. Babin and G. Laporte. Improvements to the or-opt heuristic for the symmetric traveling salesman problem. *Cahier du GERAD*, 2005.

[21] S. Agarwal K. Deb, A. Pratab and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[22] M. Laumanns E. Zitzler and L. Thiele. Spea2: Improving the strength pareto evolutionary algorithm. *Eurogen*, 3242(103), 2001.