# Predicting the Performance of Automatically Scalable Computation (ASC)

## Acknowledgements

I would like to extend my immense gratitude to Professor Margo Seltzer, my thesis advisor and mentor. Thank you for inspiring me in the area of systems and for tirelessly believing in me. In addition, I would like to thank my academic advisor Professor Finale Doshi-Velez for teaching and encouraging me in the area of machine learning. I would also like to thank Amos Waterland for helping me work with ASC. Special thanks to Nick, Sierra, and Dan for your incredibly helpful comments and proofreading. Personal thanks to my blockmates for cheering me on throughout this process and making my undergraduate experience so unforgettable. Finally, I would like to thank my family for supporting me unconditionally.

# Contents

## List of Figures

# 1 Introduction

## 1.1 ASC overview

The longstanding goal of automatic parallelization is to take a sequential program, run it on a machine with multiple processors, and achieve linear scaling with respect to the number of processors. To date, traditional methods are still far from achieving this goal. The Automatically Scalable Computation (ASC) architecture is a new approach that transforms autoparallelization into a machine learning problem. Given sequential code that compiles into a single threaded binary program, the goal of ASC is to automatically scale the performance of this sequential program as a function of the number of cores and amount of memory available [1].

ASC treats the program's execution as a stochastic walk through an exponentially large state space consisting of all of a processor's memory and registers. The execution of a single instruction corresponds to a single transition between states. ASC executes the program sequentially on one core and predicts the possible future states that the program might reach. ASC then deploys additional cores, each of which begins execution from a predicted future state for a fixed amount of time, and stores the resulting pair of start and end states in a state cache. When the sequential core hits a potentially predicted start state, it queries the state cache, and if its state matches any cached start state, then it jumps directly to the corresponding end state. This results in a speedup for the sequential core. With accurate predictions and optimal partitioning of future start and end states, ASC has been shown to achieve almost linear speedup as a function of the number of processors for some programs [1].

## 1.2 ASC predictions

The underyling principle behind ASC's ability to predict future states is that if we observe the state repeatedly at a given place in the program, we can discover patterns that would allow us to build a model to predict the states at that place over time. For instance, suppose a program increments a counter within a loop. At the place inside the loop where the counter increments, the state will increment consistently at the counter's location in memory. The “place” inside the loop can be defined by an instruction address, which is loaded into the instruction pointer (IP) during execution. We might build multiple models for multiple values of the IP. ASC harnesses this idea to predict patterns that are much more complicated and useful than the incrementing of a simple counter.

For a single IP value, ASC models the change in states over time at that IP value. ASC treats this prediction problem as an on-line learning problem: each time ASC observes a new state at the given IP, ASC gets another data point to improve its model for the states at that IP. Thus, the model improves dynamically over time. This is different from batch learning, where all of the data used for building a model is available at once. In machine learning, a number of methods have been developed to optimally approach on-line learning, including logistic regression and neural networks [2]. A given implementation of ASC can use any one of these methods.

For any machine learning method that ASC implements for on-line learning, we can assess the accuracy of the model over time. Each time ASC encounters the given IP value, it makes a prediction for the state the next time the IP is observed. We can then compare the predicted state with the actual state reached and calculate an error function value based on the accuracy of the prediction. A “good” IP value will have the characteristic that the collection of states for that IP value exhibit some kind of underlying statistical pattern, and the on-line learning method that ASC implements should be able to capture this pattern. In a perfect world, we thus expect that the accuracy of the predictions increases over time as ASC’s on-line learning model improves. However, in practice, for most IP values, the states might simply not exhibit a predictable pattern. Even if they do, ASC’s implementation of the on-line learning model will not always perfectly capture it.

## 1.3 IP selection

Since different IP values within a program can exhibit different underlying patterns of states, we face a central problem: how do we select the best IP values to model for a given implementation of ASC? The most naive approach (and current default approach) is to have the user manually supply the IP value in the form of an address when they run a program with ASC. This relies on the user’s imperfect knowledge of both the behavior of the program and the behavior of ASC's machine learning implementation. Even if the user has some heuristics to estimate the best places in the program to model the states, ASC's actual learning efficiency can be highly variable. In an ideal world, a perfect implementation of ASC would not require the user to supply an IP value at all. Instead, the user would only have to supply a program, and ASC would automatically choose an IP value to efficiently produce an accurate prediction model.

One method that is currently being developed to achieve this holy grail of automatic IP selection is a Gaussian process recognizer trained on a single program. This recognizer first runs ASC with all of the program's possible IP values for a small, fixed amount of time. This generates a set of preliminary performance metrics for ASC's internal machine learning model for each IP value. A Gaussian process regressor then takes these metrics as training data to predict future performance metrics for each IP value. The IP values that yield the highest predicted performance are then run again with ASC, and the Gaussian process regressor collects even more training data for the next iteration. Eventually, the recognizer decides on a single IP value with optimal predicted performance.

This Gaussian process recognizer has shown promising results. However, it is limited in that it must be re-trained for each program. This means that a user still could not present ASC with a completely new program and expect ASC to automatically select the optimal IP without an extra amount of training. Another limitation of this approach is that the Gaussian process recognizer does not easily share statistical strength between programs. This is important because the types of IP values that yield the best ASC performance might be common between programs or linked to the features of the program itself. If the Gaussian process recognizer is independently trained for each program, it loses the strength of these patterns.

## 1.4 Primary Goals

Given the limitations of the Gaussian process recognizer and manual IP selection, we propose an alternate approach. Our approach relies on the previous observation that for a good IP value, the error function of ASC's internal machine learning model decreases as ASC observes more states at that IP value. We will refer to each time ASC observes a new state as a *round* of training for ASC's internal machine learning model. For a given program and IP value, we build a predictive model for the error function value over the number of rounds of ASC's training. The inputs are the programs and IP values in the form of a feature space, and the target is the error function value for a given round of ASC's training.

Unlike the Gaussian process recognizer, we incorporate into the training set a wide variety of programs and their IP values. Instead of re-training for every new program, the machine learning system presented here is trained ahead of time on the IP values of many different programs, ideally capturing as much variation in the training set as possible. For a given program and IP value pair, we combine the features of both the program and IP value into a single feature set. This way, our model can capture the patterns of IP value performance that exist between programs as well as within programs. Also, unlike the Gaussian process recognizer, our machine learning system can predict the performance of a new program and IP value without ever having to run the program with ASC.

Another gain from our model is the ability to predict not only which IP values perform well with ASC, but also which programs have the potential to perform well. After all, some programs exhibit behavior that is simply not predictable by ASC at all, regardless of the IP value selected. An example of such a "bad" program would be one that exhibits true randomness, such that each state does not deterministically relate to the last. As a user, it is powerful to be able to determine the predictability of a program ahead of time without ever having to waste time attempting to run the program with ASC. Currently, researchers are working on defining the theoretical class of programs that should see performance improvements with ASC. However, for a user, it can be difficult to assess the theoretical properties of a given program in practice. The vision of our machine learning system is that a user provides a program and some of its possible IP values, and our system quickly determines whether any of these IP values have potential for speedup under ASC's current implementation. Thus, the machine learning system presented here is designed to be used in conjunction with the theoretical findings to estimate the predictability of a given program with ASC in practice.

Of course, there are many factors other than the quality of ASC's future state predictions (marked by the error function value) that affect the ultimate performance improvement that a program can achieve with ASC. For example, ASC must be able to efficiently allocate cores across a distribution of predicted states and efficiently manage the cache. However, these problems are mostly implementation dependent given accurate predictions. They would also introduce confounding variables and complexity to a machine learning model for little gain, since ideally ASC's performance improvement should correlate directly with the accuracy and partitioning of predictions. Thus, studying the effects of these factors is outside of the scope of this project.

We present ways to extract features from a program and IP value that can be used in a machine learning framework to predict ASC performance. We then present three different machine learning models for predicting ASC's error function values over time. The three machine learning models that we evaluate are Bayesian ridge regression, Gaussian process regression, and multi-task Lasso regression. We analyze the performance of these models relative to each other. In order to evaluate these machine learning models, we also present our own data set consisting of a variety of programs that exhibit different amounts of performance improvement with ASC.

In Section 2, we provide background on ASC's internal approach to automatic parallelization using machine learning. We also provide background on the machine learning methods that we use in this project to predict ASC's error function values for each program and IP value. These machine learning methods include linear regression, Gaussian process regression, and multi-task regression, and each has its relative advantages and disadvantages.

In Section 3, we present related work in the area of performance prediction for automatic parallelization. Research on performance prediction exists for many established methods of automatic parallelization beyond ASC.

In Section 4, we describe our methods of data collection and feature extraction. We build our own data set from a variety of programs, and we describe the unique qualities of those programs. We also describe the features that we extract from each program and the IP value, and the reasoning behind the selection of those features. Finally, we give a detailed description of the the machine learning models used in this project. These models include Bayesian ridge regression, Gaussian process regression, and multi-task Lasso regression.

In Section 5, we present the results of experiments run with different machine learning models. First, we present the results from the simplest Bayesian ridge regression model.

Then, with a baseline established from the Bayesian ridge regression model, we present the results of the more complex Gaussian process and multi-task Lasso regression models. We found that in general, the Gaussian process performed the best when tested on a program that it had never seen before in training.

In Section 6, we discuss the implications of the results for the development of ASC. We also propose machine learning models that can be implemented in the future to build an even better framework for ASC performance prediction.

## 2 Background

### 2.1 ASC's learning model

Before discussing the machine learning methods that can be used to estimate a program's predictability with ASC, we first provide a more detailed description of ASC's internal prediction methods. We focus on the neural network implementation of ASC, as this has been shown to produce the best on-line learning performance so far [3].

After observing a sequence of state vectors at a given IP value, ASC's neural network learns a probability distribution for the next state at that IP value. To make concrete predictions, ASC draws a set of likely future states from this probability distribution. When ASC observes the actual next state vector at the given IP value, ASC then compares the actual observed state vector to its predictions and updates the parameters of its probability distribution accordingly. We will refer to each time ASC observes a new state as a *round* of training of ASC's neural network.

The neural network implementation of ASC uses a *cross entropy loss* function to evaluate the probability distribution that it learns over time. The cross entropy loss function is a metric for evaluating the accuracy of the predicted probability distribution. Specifically, it is related the likelihood that the actual observed state is reached given the predicted probability distribution (even more specifically, it is the negative log of the likelihood). Minimizing the cross entropy loss function is equivalent to maximizing the likelihood of reaching the observed states under the predicted probability distribution.

For a “good” IP value and program, as ASC observes an increasing number of states, the predicted probability distributions for subsequent states should move closer and closer to some true underlying distribution. As this happens, the cross entropy loss value should decrease as the number of observed states increases and ASC’s prediction of future states improves.

Unfortunately, not all IP values and programs exhibit the same decrease in cross entropy loss values over time, since not all IP values have states that exhibit a consistent underlying statistical pattern over time. Thus, when evaluating an IP value’s predictability with ASC, we can examine how the cross entropy loss value changes as ASC encounters and trains on an IP value for more rounds. The target of our machine learning model is the cross entropy loss value. Our goal is to predict the cross entropy loss value at a given round of ASC’s training for a given IP value and program.

## 2.2 Regression models

The target domain for the machine learning problem is continuous, since the cross entropy loss can have any real numbered value. The feature space of programs and IPs can be continuous. Thus, regression models are well-suited for predicting the cross entropy loss values.

### 2.2.1 Linear regression

In linear regression, one of the most widely used models for supervised learning, the target values are modeled as a linear function of the inputs. A linear regression produces a set of parameters that defines the linear function mapping the inputs to outputs. For this reason, a linear regression is known as a *parametric* model [4].

One important aspect of linear regression is properly handling outliers and noisy data. With noisy data, choosing the parameters that best model the training data can result in overfitting. To mitigate this, we can place a Gaussian prior on the parameters of the linear regression. This effectively penalizes the complexity of the output function and discourages overfitting. This technique is known as *ridge regression* [4]. Ridge regression is useful for the machine learning problem presented here since different programs and IP values can exhibit a wide and noisy range of cross entropy loss values with ASC over time.

### 2.2.2 Gaussian process model

While a linear regression can parametrically model many relationships between inputs and outputs, it is limited to expressing only linear combinations of the input basis functions. A Gaussian process model, on the other hand, is a non-parametric model that can more flexibly capture an underlying non-linear function between inputs and outputs. Rather than optimizing over a parametric representation of a function, a Gaussian process model tries to model the function itself. This function is represented by its own function values at a finite but arbitrary set of points. A Gaussian process model assumes that if two inputs are similar, then the underlying function applied to those two inputs should produce similar outputs as well [4]. The flexibility of the non-parametric Gaussian process model could be useful for modeling the complex, non-linear relationship between program features and cross entropy loss values produced by ASC.

## 2.3 Multi-task regression

In the models presented above, we assume that the regression problem produces a single output for a single input. That is, for a single program, IP value, and round of ASC training, the regression model outputs a single cross entropy loss. However, this does not explicitly account for the fact that for a fixed program and IP value, there is an underlying function mapping the number of rounds of ASC training to the cross entropy loss. Thus, it can instead be useful to create a regression model that maps multiple inputs to multiple outputs. This regression model could take a fixed number of inputs from different rounds of ASC training on the same program and IP, and output the same fixed number of cross entropy loss values (one for each round of ASC training). The main advantage of a multi-task regression is that the model can share statistical strength within sets of inputs that are known to be related [5].

One disadvantage of a multi-task regression when applied to ASC is that it produces only a fixed number of outputs for a given input. Thus, for a given IP value and program, the multi-task regression can predict cross entropy loss values only for a fixed number of ASC training rounds. The linear regression and Gaussian process models, on the other hand, can predict cross entropy loss values at arbitrarily late rounds of ASC training. This limitation is significant because when predicting whether or not an IP value and program will ever reach performance improvements with ASC, it is important to consider the asymptotic behavior of the cross entropy loss. A low cross entropy loss after many rounds of ASC training would show that eventually, ASC can learn to make good predictions for a given IP value and program. Still, the predictive advantages of the multi-task regression may outweigh this limitation.

# 3 Related Work

## 3.1 Performance prediction on distributed systems

Performance prediction for parallel programs has been widely researched in the area of distributed systems. Performance metrics of interest for prediction include the program's scalability and memory and cache efficiency on a given distributed platform. Many methods exist to model the performance of programs running directly on distributed platforms. One of the most common methods is to use simulations to estimate the scalability of programs [6, 7]. These simulations are often run on protoype systems, which may not capture all of the properties of the full distributed system. Another method is to directly use a program's features to analytically and deterministically model scalability [8]. This method is limited by the fact that parallel execution can be complicated for irregular and dynamic applications, and deterministic models may not fully capture the behavior of these types of programs. A third method, as done in this thesis, is to use machine learning to predict the parallel performance of programs [9]. Machine learning methods often use data from simulations as training data to construct statistical models of program performance. Thus, they are not independent of simulations, but can perform better than simulations or analytical models alone [10].

Predicting the performance of a program on a distributed system is useful in two reciprocal ways: it helps users tune programs to be more efficient for a given system, and also makes it possible to tune the system to run optimally for a given program. Both of these purposes align strongly with the purposes of ASC performance prediction, where ASC can be thought of as analogous to the system running the program. For the distributed system, machine learning based performance prediction helps the system optimize task partitioning and scheduling [11]. Similarly, performance prediction could also helps ASC optimally run programs, though through IP selection instead. In addition, the prediction model must be interpretable in order to help users improve the design of their programs. In particular, the analytical model for distributed system performance prediction has been useful for helping programmers evaluate program design tradeoffs [12]. The machine learning models, on the other hand, tend to perform better but are more difficult to interpret. Thus, building an interpretable machine learning model for performance prediction is important to both ASC performance prediction and to general performance prediction across distributed platforms.

## 3.2 Performance prediction on compilers

While ASC can be analogous to a system on which to run programs, ASC is also a tool for autoparallelization. Aside from ASC, the more traditional approach to autoparallelization is via compilation. Compilers can parallelize programs through vectorization, loop tiling, thread-level parallelism, and other optimization techniques [13]. Currently, a growing effort is being made to predict the performance improvement of a program after using a given compiler optimization method. For instance, machine learning has successfully helped predict the performance of programs under different schemes of vectorization [14, 15]. With these predictive models, users can then select the best vectorization scheme to optimize a given program.

Beyond vectorization, machine learning has also been used to predict the performance of a program under entire optimization sequences combined to produce autoparallelism [13, 16]. These sequences include layers of vectorization and thread level parallelism that interact on a given architecture. In many cases, the compiler performs these optimizations using static analysis, which means that programs with well-defined and regular loop behavior tend to perform the best. Performance prediction is significant for compilers because a good prediction model can help users select the best compiler autoparallelization method for their program.

In fact, beyond compiler autoparallelization, performance prediction can also help users select the best general autoparallelization method for their program. This includes ASC as an autoparallelization method, as well as binary parallelization and hardware parallelization. These methods interact differently with different types of programs, yet this type of overall performance prediction has not yet been explored. In this thesis, the ASC performance predictor uses machine learning methods that could easily be generalized to analyze other autoparallelization methods as well. Thus, we take a step towards overall performance prediction.

## 3.3 Feature representation

For both distributed systems and compilers, much work has also been done on feature selection and representation of a program for predictive modeling [17, 18]. Many of these features are generated through both static and dynamic analysis, as we do here. In general, many of the program features that are useful for performance prediction on distributed systems are also useful on compilers. However, since the problem spaces are slightly different, the best feature representation for a program also varies. In some cases, researchers have collected features from the program as a whole, and in other cases, it has proven more effective to collect them on a per loop basis [14]. In the case of the ASC performance predictor, we collect features from the program as a whole, but we add the extra dimension of collecting features from the IP value as well. Feature representation for an IP value has not been widely developed and may be useful for compiler and distributed system performance prediction as well.

Beyond feature representation, the exploration of the machine learning models themselves has not been very extensive for either performance prediction on distributed systems or compilers. For the groups that used machine learning to build their predictors, they presented only a single machine learning method, and did not evaluate it in comparison to any other methods [10]. Granted, the machine learning methods varied between research groups, but we cannot really compare these methods when they are run on different problem spaces. In this thesis, we evaluate a variety of machine learning techniques for ASC performance prediction for both accuracy and interpretability. Thus, we come closer to finding the best method of predicting the performance of a program and IP value on ASC.

# 4 Experimental Methods

## 4.1 Data Collection

To solve the machine learning problem of predicting ASC's cross entropy loss values for a given IP value and program, we require a data set with two important qualities:

1. The data set must contain a variety of programs.
2. The data set must contain features that have some meaningful relation to the "predictability" of each program and IP value.

### 4.1.1 Collection of programs

For the programs in our data set, we collected a set of C programs that exhibit varying qualities. These programs vary in their cache usage patterns, loop structure, and call graph structure.

The inputs to each program are all variable. In other words, for each program, a user can manually vary the input arguments which may cause the program to follow a slightly different execution path on each run. This is important because programs that do not vary according to input are trivial for ASC to predict - if a program follows the exact same execution path through the exact same states each time it runs, then ASC will eventually be able to learn these exact states and make perfect predictions for all IP values of the program. In practice, ASC should be able to make accurate predictions for a program and IP value even when input varies. Figure 1 enumerates the programs collected and provides a short description of each program and its variable inputs.

Description of all programs collected

| Program number | Description | Variable inputs |
|---|---|---|
| 1 | Pseudo random number generator: generates an array of random integers using rand(). | 1. Size of the array |
| 2 | Modular division: divides a large number passed in as a string using the % operation. | 1. Dividend string<br>2. Integer divisor |
| 3 | Factoring: factors a large number. | 1. Integer to factor |
| 4 | 3-subset-sum: Given a set of integers, determines whether there exists a subset of 3 integers that sum to 0. | 1. Size of the set of integers<br>2. Elements in the set of integers |
| 5 | Collatz: simulates the Collatz sequence. | 1. Starting integer |
| 6 | Loop: runs an empty loop. | 1. Number of times to run the loop |
| 7 | Warshall: runs the Floyd-Warshall algorithm on a graph represented as a global array. | 1. Elements in the graph |

**Figure 1:** Complete list of all programs in the data set. The third column lists the inputs that the user can vary for that program. For the rest of the paper, we will reference the programs by the program numbers given in this table.

An important limitation on these programs is that ASC currently runs stably with C programs using a specially optimized version of libc called diet libc. Optimized for small size, diet libc creates small statically linked binaries for C programs. This means that our data set is limited to C programs that do not have bulkier dependencies such as the containers in the C++ standard library. Still, even with this limitation on libc, we can manually create programs that capture a high level of complexity and variability. Thus, we can still capture the qualities of predictable programs and IP values with ASC using this data set.

Another limitation to these programs is that ASC will not run any programs that dynamically allocate memory on the heap (this is because it has not implemented certain system calls such as sbrk). Thus, for now, we cannot analyze the performance of ASC with varying heap access patterns. However, we can still vary memory access and allocation on the stack and capture different memory access patterns this way.

### 4.1.2 Feature extraction

To analyze each program and IP value, we extract features using both static and dynamic analysis. Our main goal with feature extraction is to extract sufficiently many important features that have some meaningful relation to the "predictability" of the program and IP. We extract features for both the program as a whole and for each individual IP value within the program.

We extract many important features from the program and IP values without ever executing the program. These static features can come from the program binary, assembly, and the program text itself. We also extract many useful features from an execution of the program. By running the program, we can observe cache usage, the function call graph, and actual numbers of times that a program encounters a given IP value.

#### Compilation

Many of the features that we extract depend on the program being compiled. We compile all of the programs in the same way, using the gcc compiler with the following flags:

*-nostdinc -isystem /usr/include/diet -ggdb3 -static -nostdlib -L /usr/lib/diet/lib -lc -lm*

One important effect of these flags is that they substitute the standard C library used by these programs with diet libc (described in the previous section).

#### Feature summary

We present a summary of all of the features extracted in the following table. We divide our features into subsets, each of which we describe in more detail in the upcoming sections. In total, we extracted 304 features for each input.

All input features (304 features)

| Feature set | Description |
|---|---|
| 1 | Program number |
| 2 | IP value |
| 3 | ASC training rounds (5 features) |
| 4 | State vectors (1 feature) |
| 5 | Program binary (256 features) |
| 6 | Program source (3 features) |
| 7 | Call graph (10 features) |
| 8 | Cache usage (13 features) |
| 9 | Program assembly (12 features) |
| 10 | Instruction reads (2 features) |

**Figure 2:** Complete list of all features collected, divided into categories of feature sets. Feature sets 3-4 vary at the level of each round of training of ASC on a given program and IP. Feature sets 5-8 vary for each program as a whole. Feature sets 9-10 vary for each IP value within a program.

### ASC training rounds

One important feature that can impact ASC's cross entropy loss over time is the number of rounds that ASC is allowed to train on each input. As mentioned earlier, the programs in this data set all have variable inputs, such that the execution path of the program varies slightly for each input. For a given program and IP value, ASC trains the same neural network across multiple runs of the program on multiple inputs. If ASC trains its neural network for too many rounds on a single input, then the neural network may overfit to the execution path of that single input. On the other hand, if ASC is not allowed enough rounds of training on a single input, then it may never learn anything meaningful about the program execution pattern for a given input. Thus, the number of rounds that ASC trains on a given input is important to take into account when predicting the performance of ASC's neural network.

For a given input, the user manually specifies the maximum number of rounds that ASC is allowed to train on that input. When building our data set, we manually vary the number of rounds of training allowed for each input for different programs and IPs. We then record these numbers of rounds of training as features. Note that there is a difference between the number of times that we run ASC on a given input and the number of rounds of training that ASC gets on that input. If we run ASC on the same input 2 times with 10 rounds of training each time, then the total number of rounds of training on that input is 20 rounds.

ASC training rounds (5 features)

| Feature number | Description |
|---|---|
| 1 | Overall round number: ASC's current round of training taking into account all runs on all inputs. |
| 2 | Input number: the number of different inputs we have run ASC on. If we ran ASC on the program and IP with 3 different inputs, then input number ranges from 1 to 3. |
| 3 | Run number: the number of times we have run ASC on the current input. If we ran ASC with 2 runs per input, then the run number ranges from 1 to 2. |
| 4 | Total rounds allowed: total number of rounds of training allowed on the current run of the current input. |
| 5 | Current round number: ASC's current round of training within the current run on the current input. |

**Figure 3:** Complete list of features related to the number of rounds that ASC is allowed to train on a given input.

**State vectors**

Perhaps the features most directly related to the ASC predictability of a program and IP value are the features of the state vectors themselves during program execution. Each time the program encounters a given IP value during execution, we observe the entire state of the program in a state vector. We do this using ASC's own infrastructure. Each round that the program encounters the given IP value, we record the Hamming distance between the current state vector and the previous state vector from the last time the program encountered the given IP value.

Ideally, the features that we extract should not depend on actually running the program with ASC. State vector features still fall into this category, since it is entirely possible to develop a tool independent of ASC that can observe the state vectors of a program at a given IP value.

State vectors (1 feature)

| Feature number | Description |
|---|---|
| 1 | Hamming distance between the state vector on the current round and the state vector on the last round. |

**Figure 4:** Complete list of features related to the state vectors obtained each time the program encounters a given IP value.

**Program binary**

The program binary itself can also contain useful static features of the program as a whole. Malware detection is one area that significantly relies on analysis of the program binary [19]. In malware detection, N-gram features extracted from the binary have successfully helped classify programs as malware. Thus, these N-gram features can reveal non-trivial information about the program, and may be useful for analyzing the way the program will behave with ASC as well. In this project, we use hexdump to extract the 1-gram features from the binary. In other words, for each program, we record the count of the number of occurrences of each byte in the hexdump. This yields 256 features for each program.

Program binary (256 features)

| Feature number | Description |
|---|---|
| 1 - 256 | Number of occurances of each 1-gram from the hexdump output. |

**Figure 5:** Complete list of features related to the program binary.

**Program source**

By manually parsing the program source code, we can extract some simple static features including the number of lines of code, number of words in the code (not including comments), and number of bytes in the source code file. These are of course fairly rough metrics when it comes to the program's execution - after all, programs can vary greatly in their source code but still execute in the same way. However, even given that variability, these metrics may still roughly correlate with the complexity of the program.

Program source (3 features)

| Feature number | Description |
|---|---|
| 1 | Number of lines of code. |
| 2 | Number of words of code, including comments. |
| 3 | Number of bytes in the source file. |

**Figure 6:** Complete list of features related to the program's source code.

**Function call graph**

The function call graph is also an important source of information for the predictability of a program. With the function call graph, we can observe any recursive or cyclical behavior within a program. We can also pinpoint the functions in which the program spends most of its time. To analyze the function call graph, we use a tool called gprof. gprof is a profiling tool that is part of GNU Binutils.

Call graph (10 features)

| Feature number | Description |
|---|---|
| 1 | Total number of functions declared. |
| 2 | Total number of function calls made. |
| 3 | Total program runtime. |
| 4 | Highest percentage of total runtime taken by a single function. |
| 5 | Highest percentage of function calls taken by a single function. |
| 6 | Variance of the runtimes taken by each function. |
| 7 | Variance of the number of calls taken by each function. |
| 8 | Maximum number of parents for any function in the call graph. |
| 9 | Maximum number of children for any function in the call graph. |
| 10 | Total number of recursive calls made. |

**Figure 7:** Complete list of features related to the program's function call graph.

**Cache usage**

One important set of features that we get from running the program is information about the program's processor cache usage. To observe this, we use the callgrind tool. Callgrind is a profiling tool for C programs that is part of the Valgrind framework. It can analyze the usage of the Level 1 (L1) and Level 2 (L2) processor caches when running a C program. On most architectures, when a program accesses memory, the processor checks the L1 cache first, followed by the L2 cache.

The L1 and L2 cache store both instructions to be executed by the processor and data to be written to primary memory. Each time the processor executes an instruction, it first checks if the instruction located at the IP value is already in the L1 instruction cache. The program will only fetch the instruction from primary memory if it is not already in the L1 instruction cache. Instruction cache usage can improve efficiency in loops, since if the program encounters a given IP value many times, the instruction it refers to can quickly be queried from the L1 instruction cache. Likewise, if the program frequently accesses certain pieces of data in primary memory, the L1 and L2 caches can improve efficiency if they hold those frequently accessed pieces of data.

The hit rates of the L1 and L2 caches can give a useful profile of the program's memory usage patterns. In particular, from the point of view of ASC, programs with high cache hit rates may exhibit more regular memory usage patterns and more predictable states. One important caveat is that the cache hit rates depend on the cache eviction policies implemented by the L1 and L2 caches. Thus, the implications of the cache hit rates for memory usage patterns and state predictability may vary for different cache implementations. In this project, we run all programs on the same machine using the same processor caches, so this variation does not occur within our data set. However, this would be an important factor to keep in mind when building future data sets.

Cache usage (13 features)

| Feature number | Description |
|---|---|
| 1 | Total number of L1 cache instruction reads. |
| 2 | Total number of L1 cache data reads. |
| 3 | Total number of L1 cache data writes. |
| 4 | L1 cache instruction read misses. |
| 5 | L1 cache data read misses. |
| 6 | L1 cache data write misses. |
| 7 | L2 cache instruction read misses. |
| 8 | L2 cache data read misses. |
| 9 | L2 cache data write misses. |
| 10 | L1 cache instruction read miss rate. |
| 11 | L1 cache data read miss rate. |
| 12 | L1 cache data write miss rate. |
| 13 | L2 cache data and instruction miss rate. |

**Figure 8:** Complete list of features related to the program's cache usage.

**Program assembly**

Beyond the program source code, the assembly code provides more detailed information about each IP value of the program. We disassemble the programs collected here using objdump. Objdump is a tool which is part of GNU Binutils that can disassemble executable files. We obtain the assembly code for each program by using objdump to disassemble each compiled binary of each program.

The assembly code of a program can be slightly less variable when it comes to program behavior than the program text. However, the assembly code for a given program can vary based on the compiler. This is not a problem in this project because we compile all of our programs in the same way. However, this may be an important consideration to make when building future data sets.

Program assembly (12 features)

| Feature number | Description |
|---|---|
| 1 - 10 | Binary features indicating instruction type of a given IP value (e.g. jmp, call, mov, lea, cmp, inc, mul, add, or, push). |
| 11 | Whether or not the IP value is the target of a jump instruction. |
| 12 | Distance of the given IP value from the target of a jump instruction (i.e. the number of instructions between the current IP value and the last target of a jump instruction). |

**Figure 9:** Complete list of features related to program assembly code.

**Instruction reads**

In addition to cache hit rates, callgrind can also get the number of times that a given instruction is read by the program. In other words, callgrind can output the number of times that a given address was loaded into the instruction pointer. We will refer to this number as the “Ir” number, or number of “instruction reads” for a given IP value. For each

IP value, callgrind outputs an Ir value. This Ir value is a particularly important feature associated with the IP value, since the Ir value represents the frequency with which the program encounters the given IP value. If this frequency is high, then the IP value may be a promising place where the program exhibits some predictable state. On the other hand, if the frequency is too high (e.g. if the IP value is hit every other cycle), then perhaps the instruction is at a trivial part the program's execution, and the states may still not exhibit any appreciable pattern.

Instruction reads (2 features)

| Feature number | Description |
|---|---|
| 1 | Number of instruction reads for a given IP value. |
| 2 | Percentage of total instruction reads attributed to the given IP value. |

**Figure 10:** Complete list of features related to instruction reads.

### 4.1.3 Collection of target cross entropy loss values

In addition to extracting all of the features above, we also collected the cross entropy loss values that ASC achieved when running each program and IP value. These cross entropy loss values form the targets of our machine learning problem. To collect these, we followed the following procedure for each program:

1. We compiled the program using the method described in the previous section.
2. We extracted a list of all possible IP values for the program by parsing the objdump assembly code output for the program.
3. For each possible IP value, we ran ASC with the program and IP value. In these runs, we varied the inputs as well as the number of rounds of training that ASC was allowed to have on each input. ASC trained the same internal neural network across all runs on all inputs for the given program and IP value.
4. Each time the program encountered the IP value, ASC produced a cross entropy loss

value. If ASC encountered the IP value a reasonable number of times, then we saved the cross entropy loss values for the current IP value. The number we assumed was "reasonable" in this project was 5 encounters of the IP value, since ASC's cross entropy loss did not seem to change appreciably if it encountered an IP value fewer than 5 times.

## 4.2 Machine learning models

### 4.2.1 Notation

Throughout the rest of this paper, we use the following general notation:

- Bold variables (e.g. $\mathbf{x}$) indicate vectors and matrices with multiple elements, whereas variables not in bold indicate scalar values (e.g. $x$).
- Variables with a "hat" represent values estimated by the machine learning models, whereas variables without the hat represent the "true" values of those variables. For instance, $\mathbf{y}$ represents the actual given set of target values, while $\hat{\mathbf{y}}$ represents the predicted target values inferred by a machine learning model.
- $\mathcal{N}(\mathbf{x}, \sigma^2)$: probability density function of a normal (Gaussian) distribution with mean $\mathbf{x}$ and variance $\sigma^2$.
- $\mathcal{N}(\mathbf{y}|\mathbf{x}, \sigma^2)$: probability density function of a normal (Gaussian) distribution with mean $\mathbf{x}$ and variance $\sigma^2$ evaluated at $y$.
- $p(\mathbf{x}|\theta)$: the probability that some data values $\mathbf{x}$ occur given some parameters $\theta$.
- $E[\mathbf{x}]$: the expected value of a random variable $\mathbf{x}$.

We also use the following variables to symbolize the following values:

- $N$: total number of data points (one for each program, IP value, and round number).
- $D$: total number of features for each data point.
- $\mathbf{y}$: target vector: a vector of length $N$ containing a cross entropy loss for each program, IP value, and round number.

- $\mathbf{x}$: matrix of input vectors: $N$ by $D$ matrix containing the full set of features for all programs, IP values, and round numbers. The $i$th row of $\mathbf{x}$ is denoted $\mathbf{x}_i$.

### 4.2.2 Bayesian ridge regression

The first model that we describe and evaluate is a model for linear regression. As mentioned in the Background section, in linear regression, the target values are modeled as a simple linear function of the inputs. While we know that the target cross entropy error value is most likely not a simple linear function of the features of the program, a linear regression is still useful as a baseline that we can use to evaluate more complicated models, such as the Gaussian process model. Linear regression can also still generate a decent estimate of the evolution of the cross entropy loss values over rounds of ASC's training, even if it will not be a perfect fit.

In our data set, different programs and IP values can exhibit a wide and noisy range of cross entropy loss values with ASC over time. Thus, it is important that we handle outliers with care. To decrease the influence of outliers, we use ridge regression. In ridge regression, we place a Gaussian prior on the parameters of the linear regression. This effectively penalizes the complexity of the output function and discourages overfitting.

The goal of linear regression is to learn a set of weights $\hat{\mathbf{w}}$ such that we can infer each predicted target value $\hat{y_i}$ by computing

$$\hat{y_i} = \hat{\mathbf{w}}^\top \mathbf{x}_i$$

In Bayesian ridge regression, we actually model distribution of each target $y_i$ as a Gaussian distribution as follows:

$$p(y_i|\mathbf{w}^\top \mathbf{x}_i, \sigma) = \mathcal{N}(y_i|\mathbf{w}^\top \mathbf{x}_i, \sigma^2)$$

To estimate a value for $\hat{\mathbf{w}}$, we compute the value of $\mathbf{w}$ that produces the highest maximum likelihood of the occurrence of the input data $\mathbf{x}, \mathbf{y}$. This is part of computing a maximum a posteriori (MAP) estimation. The likelihood of the input data $\mathbf{x}, \mathbf{y}$ given some parameters $\mathbf{w}, \sigma$ is given by

$$p(\mathbf{x}, \mathbf{y} | \mathbf{w}, \sigma)$$

For ease of numerical computation, we maximize the negative log likelihood instead of the likelihood. The log likelihood of the data is given by

$$\log p(\mathbf{x}, \mathbf{y} | \mathbf{w}, \sigma) = \sum_{i=1}^{N} \log p(y_i, \mathbf{x}_i, \mathbf{w}, \sigma)$$
$$\log p(\mathbf{x}, \mathbf{y} | \mathbf{w}, \sigma) = -\frac{1}{2\sigma^2} \sum_{i=1}^{N} (y_i - \mathbf{w}^\top \mathbf{x})^2 - \frac{N}{2} \log(2\pi\sigma^2)$$

To avoid overfitting, we use ridge regression to place a Gaussian prior on $\mathbf{w}$ with some standard deviation $\tau$ as follows:

$$p(\mathbf{w}) = \prod_{j} \mathcal{N}(w_j | 0, \tau^2)$$

Let $\lambda = \frac{1}{\tau^2}$. Then the final optimization equation for ridge regression is given by

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w} \in \mathbb{R}^D} \sum_{i=1}^{N} (y_i - \mathbf{w}^\top \mathbf{x})^2 + \lambda \sum_{j=1}^{D} w_j$$
$$\hat{\mathbf{w}} = (\mathbf{x}^\top \mathbf{x} + \lambda I_D)^{-1} \mathbf{x}^\top \mathbf{y}$$

This can be computed efficiently using QR decomposition [4].

The parameter $\lambda$ is also referred to as the "shrinkage parameter." The shrinkage parameter $\lambda$ controls the regularization level: as $\lambda$ approaches 0, we arrive at the simple least squares linear regression solution. As $\lambda$ increases, $\hat{\mathbf{w}}$ approaches $\mathbf{0}$.

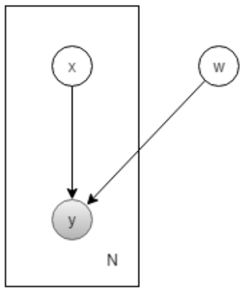

**Figure 11:** Graphical model for Bayesian ridge regression. $\mathbf{x}$ denotes a single input feature vector. $y$ denotes a single target cross entropy value. $\mathbf{w}$ denotes a weight vector. There are $N$ different feature vectors and target values.

In this project, we perform Bayesian ridge regression as described above using all of the 304 input features to build our input set $\mathbf{x}$, and using all of the cross entropy loss values as target values $\mathbf{y}$. We wrote our own implementation of the Bayesian ridge regression in Python.

### 4.2.3 Gaussian process model

If $f$ is a function from $\mathbf{x}$ to $y$, the previous Bayesian regression model tries to represent $f$ parametrically using a weight vector $\mathbf{w}$. A Gaussian process model on the other hand, makes predictions by trying to model $f$ directly [4]. The Gaussian process treats the set of all data points $\mathbf{x}_i$ as jointly Gaussian with a covariance matrix $\mathbf{K}$. The entries in the covariance matrix are given by $\mathbf{K}_{i,j} = \varphi(\mathbf{x}_i, \mathbf{x}_j)$, where $\varphi$ is a kernel function that relates two data points. If the data points $\mathbf{x}_i, \mathbf{x}_j$ are similar according to the kernel function $\varphi$, then their covariance should be low and $f(\mathbf{x}_i), f(\mathbf{x}_j)$ should be similar as well.

Let $\mathbf{f}$ be the set of all values $f(\mathbf{x}_i)$. Then for any finite set of points $\mathbf{x}$,

$$p(\mathbf{f}|\mathbf{x}) = \mathcal{N}(\mathbf{f}|\mu, \mathbf{K})$$

where $\mathbf{K}_{i,j} = \varphi(\mathbf{x}_i, \mathbf{x}_j)$ and $\mu_i = E[f(\mathbf{x}_i)]$

In this project, $\mathbf{x}_i$ is a feature vector for a given program, IP value, and round number, and $f(\mathbf{x}_i)$ is the predicted cross entropy loss for that program, IP value, and round number.

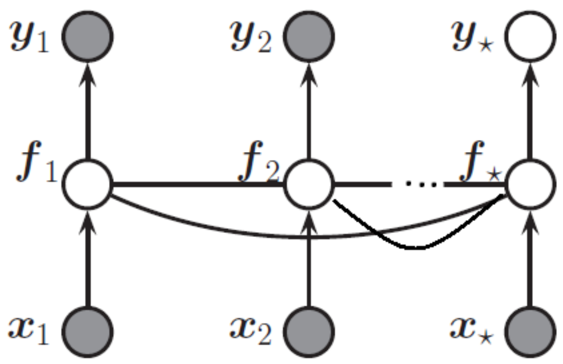

**Figure 12:** Graphical model for Gaussian process model. $\mathbf{x}_i$ denotes a single input feature vector. $y_i$ denotes a single target cross entropy value. $f_i$ denotes an underlying function mapping $\mathbf{x}_i$ to $y_i$, $f_i = f(\mathbf{x}_i)$. The undirected edges represent the covariance terms, which fully interconnect the hidden nodes. We denote an unseen data point by $\mathbf{x}_*, y_*, f_*$.

To control for overfitting and achieve convergence for the Gaussian process model, we add a "nugget" to the model. A "nugget" is a set of values that we add to the diagonal of the correlation matrix: $\text{nugget}_i$ is the scalar value that we add to $\mathbf{K}_{i,i}$. In general, $\text{nugget}_i$ should specify the variance of the noise expected for each target value $y_i$. In this project, we achieved convergence by setting the nugget to

$$\text{nugget}_i = 20 + y_i$$

In doing this, we assume that the variance for the noise for each target value is roughly correlated with the target value itself. In other words, we assume that higher target values are likely prone to more noise than lower target values. To implement our Gaussian process model, we use scikit-learn's GaussianProcess module in Python [20].

### 4.2.4 Multi-task regression

Both the Gaussian process model and the Bayesian ridge regression model described above are used to predict a single, one-dimensional target value for each input. In multi-task regression, instead of predicting a single target $\hat{y}$ for each input, we predict a set of targets $\hat{\mathbf{y}}$ for each input. Multi-task regression could be especially useful in this project for predicting cross entropy loss values over time. For each program and IP value with features $\mathbf{x}_i$, the multi-task regression would produce a vector of outputs $\hat{\mathbf{y}}$, where each entry in $\hat{\mathbf{y}}$ represents a cross entropy loss for an individual round of ASC training on that program and IP value. Intuitively, by grouping together all of the cross entropy losses for a given program and IP value, the multi-task regression can share statistical strength for data points across a given program and IP value.

It is important to note that the dimension of the output vector $\hat{\mathbf{y}}$ is fixed in multi-task regression. That is, for each input, the multi-task regression will output exactly $M$ target values, where we specify $M$ before training. This means that in this project, our multi-task regression can predict only up to $M$ rounds of ASC training for each program and IP value. This is a strong constraint in comparison to the other single task models, both of which can predict cross entropy loss values for an arbitrarily high number of rounds of training. However, even with this constraint, it is still worth investigating whether or not the multi-task regression model results in more accurate predictions than the two single-task regression models.

Many different types of models for multi-task regression exist. In this project, we use Lasso regression with mixed $l_1/l_2$ normalization. Lasso regression is a variant of linear regression. Thus, it produces an $M$ by $D$ dimensional weight matrix $\mathbf{w}$ such that

$$\hat{\mathbf{y}} = \mathbf{x}\mathbf{w}$$

where $\hat{\mathbf{y}}$ is an $M$ dimensional vector of cross entropy values for $M$ rounds of ASC training. For a single cross entropy value $y_i$,

$$\hat{y_i} = \mathbf{w}_i^\top \mathbf{x}_i$$

In order to produce $\hat{\mathbf{w}}$, we optimize the following objective function:

$$\hat{\mathbf{w}} = \arg\min_{\mathbf{w}} ||\mathbf{y} - \mathbf{x}\mathbf{w}||_2^2 + \lambda||\mathbf{w}||_{2,1}$$

where

$$||\mathbf{w}||_{2,1} = \sum_{i=1}^{M} \sqrt{\sum_{j=1}^{D} w_{ij}^2}$$

Here, $||...||_2$ denotes the $l_2$-norm, and $||...||_{2,1}$ denotes the mixed $l_1/l_2$ norms. The regularization level is given by $\lambda$.

To implement our multi-task Lasso regression model, we use scikit-learn's MultiTaskLasso module in Python [20].

For multi-task Lasso regression on our data set, we set $M = 10$. In other words, for each program and IP value, we predict cross entropy loss values for the first 10 rounds of ASC's training. We set $M = 10$ because this allows us to use the majority of the programs and IP values for which we collected data. For almost every program and IP value, we were able to run ASC for at least 10 rounds (though in many cases we were able to run ASC for many more rounds). While 10 is a fairly small number of training rounds, we found that after 10 rounds of training, it usually became clear whether the program fell in the range of good, bad, or super bad.

## 5 Results and analysis

### 5.1 Preliminary data analysis

Before analyzing the data set with any machine learning algorithms, we first provide a statistical overview of the data set as a whole.

#### 5.1.1 Distribution of target values

The following figure shows the frequency distribution of all of the target cross entropy loss values for all data points in the entire data set. This includes one cross entropy loss value for each data point, where each data point is associated with a program, IP value, and ASC training round number.

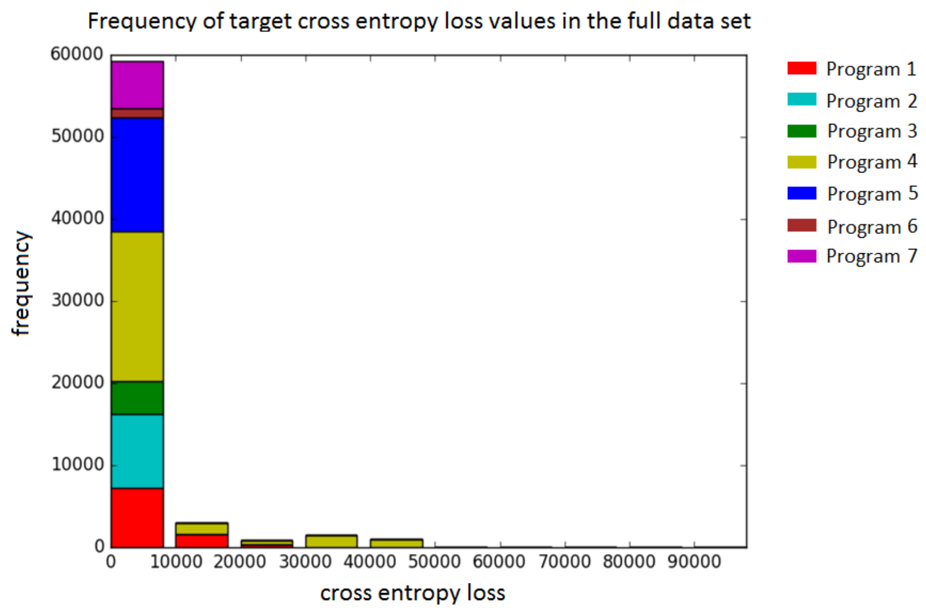

**Figure 13:** Histogram of cross entropy loss values over all data points in the full data set. The color breakdown shows the frequency contributed by each program. The mean cross entropy value is 2989.27, and the variance of the cross entropy values is $7.29 \times 10^7$

A complete histogram of the cross entropy loss values for the full data set shows that most of the cross entropy loss values for all programs fall between 0 and 20000. The high cross entropy values above 20000 form the tail of the distribution. In addition, both the mean and the variance of the target cross entropy loss values are high - the mean is 2989.27, and the variance is $7.29 \times 10^7$. *For perspective on how high these values are, note that in general, programs and IP values with cross entropy loss values greater than 100 no longer achieve any effective speedup with ASC.*

We also provide a breakdown of the cross entropy loss values for each program (Figures 14 and 15). Across all of the programs, the distributions of the cross entropy loss values are also heavily skewed right.

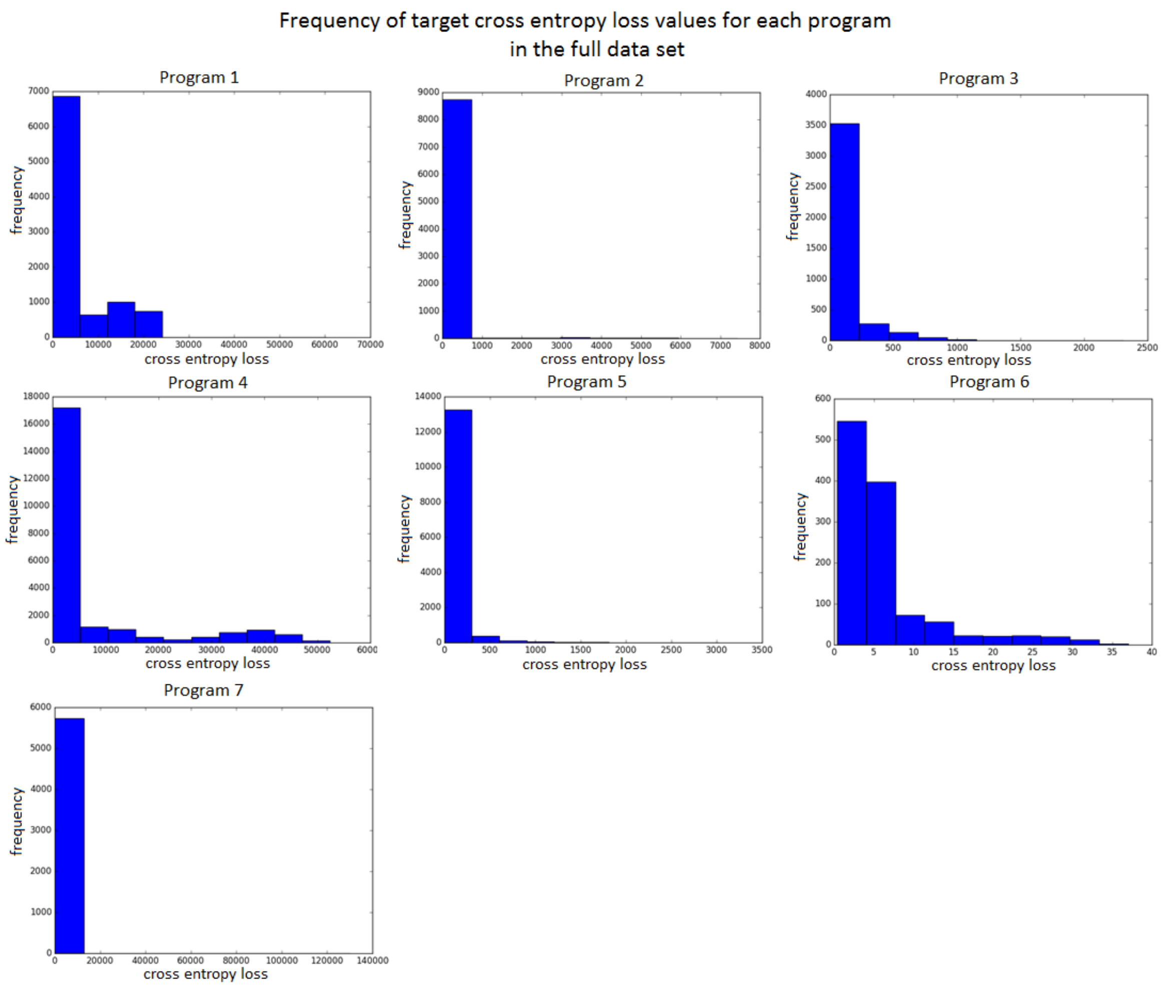

**Figure 14:** Histograms of cross entropy loss values for each program in the full data set.

Means and variances of cross entropy losses for each program in the full data set

| Program number | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Mean | 4608.03 | 125.39 | 98.44 | 6547.66 | 113.58 | 6.18 | 297.17 |
| Variance | $4.76 \times 10^7$ | $2.92 \times 10^5$ | $2.93 \times 10^4$ | $1.62 \times 10^8$ | $2.97 \times 10^4$ | 38.84 | $1.14 \times 10^7$ |

**Figure 15:** Means and variances of cross entropy losses for each program in the full data set.

For Bayesian models, we can think of the frequency distribution of the target values as a prior distribution for the model. In this data set, the heavy skewness means that a simple Gaussian prior may not perform as well as it would on a data set with a less skewed target distribution. However, it is still important to note that the target distribution is not multi-modal. This means that we do not need to use a more complicated prior than a single Gaussian distribution.

### 5.1.2 Reduced data set

The full data set contains many data points with cross entropy loss values above 20000. However, in practice, it is less important to be able to predict the extremely high cross entropy loss values at the tail of the histogram. Instead, to predict whether or not a program can achieve speedup with ASC, it is sufficient to predict the cross entropy loss values only up to some ceiling, such as 100. Thus, to help the machine learning models better model the data at reasonably low cross entropy loss values, we take out a subset of the full data set that contains only the data points with cross entropy loss values less than 100. We also take only the first 10 rounds of ASC training for each IP value in order to match this smaller data set with the one required for multi-task regression. This way, we can use the same smaller data set to compare performance across all three machine learning models. For the rest of this analysis, we will refer to the original data set as the "full data set" and this smaller subset as the "reduced data set." The full data set contains $65,717$ data points, and the reduced data set contains $5,982$ data points.

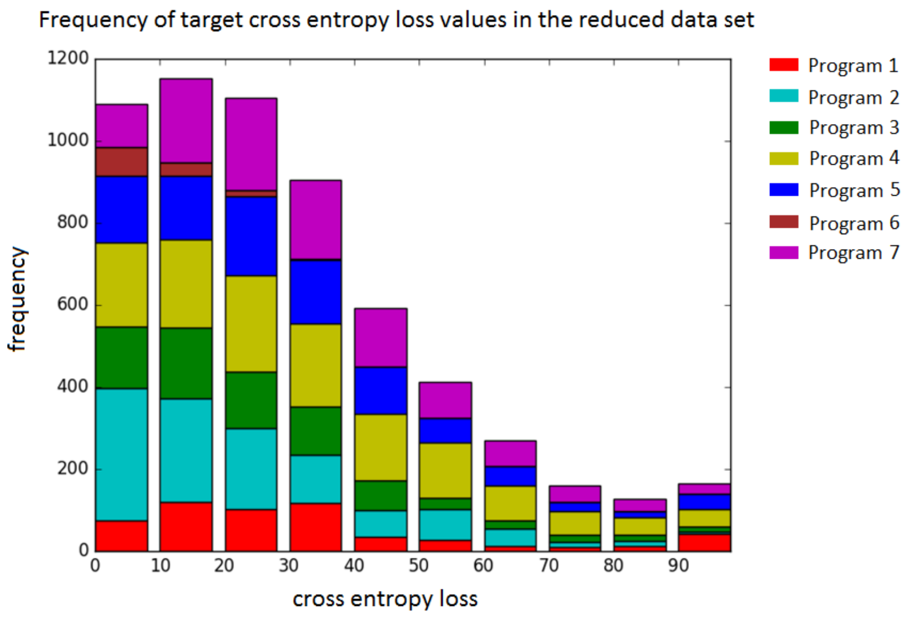

**Figure 16:** Histogram of cross entropy loss values over all data points in the reduced data set. The color breakdown shows the frequency contributed by each program. The mean cross entropy value is 31.24, and the variance of the cross entropy values is 523.37.

In the reduced data set, the frequency of the target cross entropy values is still skewed right. However, the skewness is much less dramatic (Figure 16). This is true across all of the programs individually as well (Figure 17).

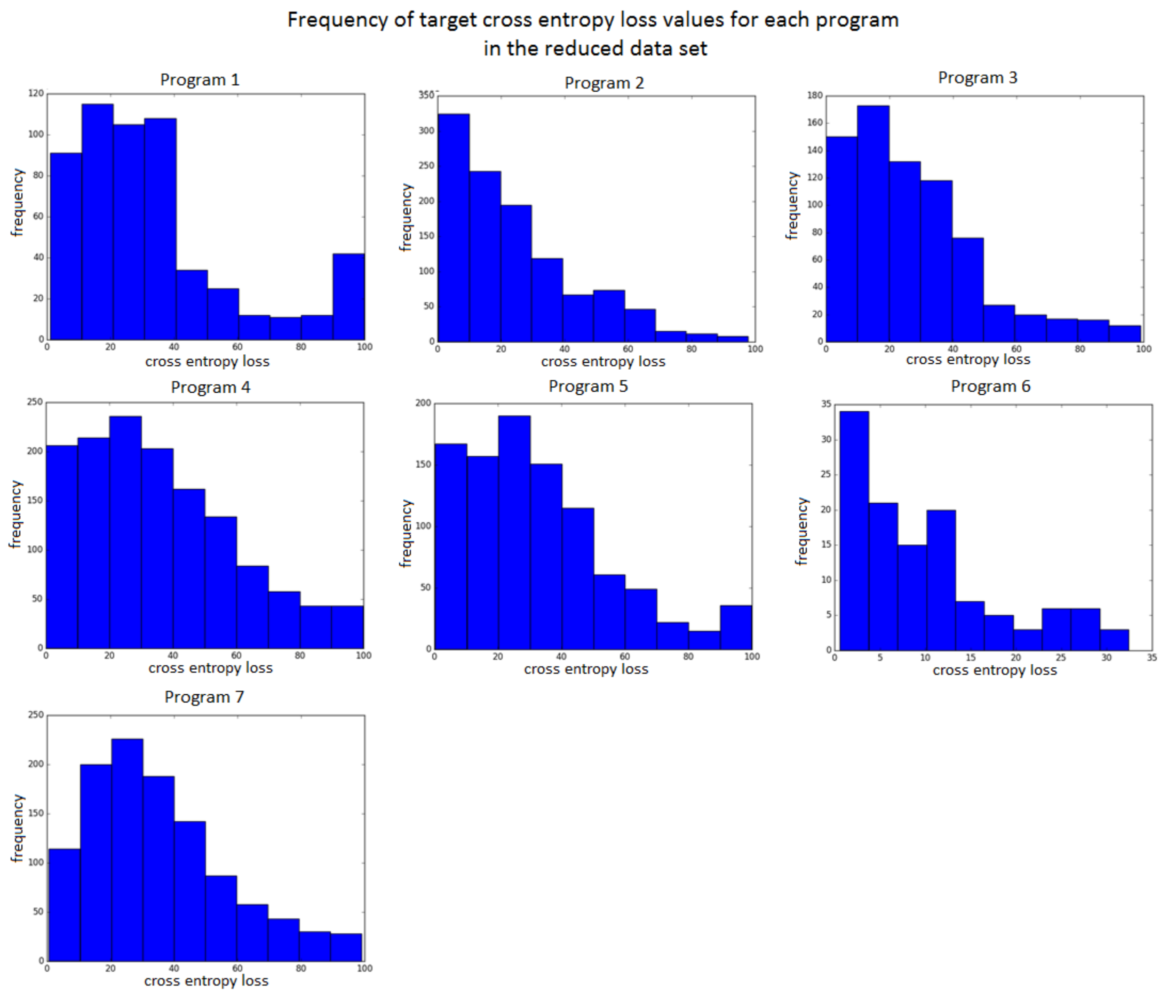

**Figure 17:** Histograms of cross entropy loss values for each program in the reduced data set.

Means and variances of cross entropy losses for each program in the reduced data set

| Program number | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| Mean | 33.35 | 24.19 | 27.48 | 35.73 | 32.57 | 10.17 | 35.23 |
| Variance | 638.68 | 391.99 | 443.93 | 580.02 | 526.17 | 66.70 | 474.35 |

**Figure 18:** Means and variances of cross entropy losses for each program in the reduced data set.

One important takeaway from looking at the program distributions individually is that each program exhibits different magnitudes of cross entropy loss values overall. For instance, the mean of the cross entropy loss values for program 6 is 10.17, whereas the mean is 35.73 for program 4. This shows that there is indeed some variability between the results that the programs produce when run with ASC. This also suggests that some programs have a higher proportion of IP values with low cross entropy loss values, and that the features of the program itself can play a role in the cross entropy loss values that we expect to see from ASC.

Even though each program produces different magnitudes of cross entropy loss values overall, we also observe that the shape of the distribution of the cross entropy values still looks similar among all of the programs (Figure 17). This is interesting because it suggests that there is some underlying similarity in the distribution of cross entropy loss values across the IP values and round numbers within each program.

## 5.2 Performance evaluation metrics

To evaluate the performance of each machine learning model, we calculate a few standard summary metrics for each model. Summary metrics include the the coefficient of determination and overall root mean squared error of the model predictions. This section provides background on some of these metrics.

### 5.2.1 k-fold cross validation

In k-fold cross validation, we split the $N$ data points randomly into $k$ different subsets. We then iteratively set each of those $k$ subsets as the test data set, and the rest of the $k-1$ subsets as the training set. We refer to the setting of one of the $k$ subsets as the test set as a "fold." For each of these $k$ folds, we evaluate the performance of the machine learning algorithm on the fold.

This technique is useful because if we just extracted a single test set at random from the data and computed evaluation metrics on that single test set, then we may be subject to sampling bias. After all, the machine learning model may perform particularly well or particularly poorly on the single small test set we sampled. On the other hand, if we evaluate the performance of the machine learning model on k different folds, we evaluate the machine learning model on a much larger sample size and achieve a less variable picture of the performance of the machine learning model on the data set as a whole.

For the following analysis of the machine learning models, we compute summary statistics for 10 folds. We partition the data set at random into 10 different folds, and for each fold, we test on one partition and train on the other nine partitions. It is important to note that we do not impose the limit that the data points in the test set all come from some specific program. Thus, the test set and training set could both contain data points from the same program. In practice, this does not simulate predicting the cross entropy loss values for programs that have not been seen before. However, the mean statistics over the 10 folds still provide a useful overall summary of the performance of the machine learning model. We also perform more detailed analysis on programs that have not been seen before for each machine learning model individually.

### 5.2.2 Root mean square error (RMSE)

One summary metric for evaluating the overall performance of each machine learning model is the root mean squared error (RMSE). For each of the $k$ folds, we compute a mean squared error (MSE) for the test set. We then average the MSEs obtained across all $k$ folds and take the square root to obtain an overall RMSE. The formula for this is as follows:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{k} MSE_i}{k}}$$

A lower RMSE generally implies a better fit for the machine learning model on the data. It is important to note that larger data sets with larger variance will tend to have a larger RMSE than smaller data sets even if the model predictions appear to fit in a relatively similar way. This is because the RMSE is simply a sum of the mean squared errors for all data points, and more data points simply results in more mean squared errors to sum. Still, the mean squared error is a useful metric for comparing model fits across the same data set.

### 5.2.3 Coefficient of determination ($r^2$)

Another useful summary metric for evaluating the overall performance of a regression model is the coefficient of determination ($r^2$). The coefficient of determination is defined as follows:

$$r^2 = 1 - \frac{u}{v}$$
$$u = \sum (y_{true} - y_{pred})^2$$
$$v = \sum (y_{true} - y_{mean})^2$$

where $u$ is defined as the regression sum of squares, and $v$ is defined as the residual sum of squares.

The coefficient of determination ranges between $-\infty$ and 1. A coefficient of determination of 1 indicates a perfect fit of the regression on the true data. The coefficient of determination will be 0 if the regression only outputs the sample mean of the data. A negative coefficient of determination indicates that the regression performed worse than the most naive approach of simply outputting the sample mean of the data. Unlike the RMSE, the coefficient of determination can be used to compare the fits of different models across different data sets.

### 5.2.4 Analysis of individual IP values

In k-fold evaluation, all of the test sets are chosen at random without consideration for which programs the data points came from. However, in practice, our goal is to be able to predict the cross entropy loss values for an individual IP value and program that we have never seen before.

To analyze the ability of a model to do this, we show the model's predictions for an isolated program and IP value. In particular, we define three different types of IP values:

1. "good": A good IP value for a program is one for which the cross entropy loss reaches values less than 20. ASC only achieves appreciable speedup if its predictions are accurate enough to reach these low cross entropy loss values.
2. "bad": For IP values with cross entropy loss values greater than 50, ASC achieves little to no speedup. We define these IP values as "bad".

3. "super bad": As shown in the full data set, some IP values yield cross entropy loss values over 10000. Not only do these not result in appreciable speedup, but they can actually cause the ASC execution of the program to be significantly slower due to large amounts of overhead.

In the upcoming analysis of each machine learning model, we will compare the performance of each machine learning model on each type of IP value. To do this, we selected three different IP values from the data set to act as representative examples of good, bad, and super bad IP values. For variety, we made sure that these IP values also came from different programs (each program has a range of good, bad, and super bad IP values). More specifically, the good IP value came from program 5, the bad IP value came from program 1, and the super bad IP value came from program 4. For both the full and reduced data set, we use the same good and bad IP values. The super bad IP value applies only to the full data set, since in the reduced data set we removed all data points with cross entropy loss over 100.

For each of the selected representative IP values, the individual plots of cross entropy loss values over ASC training rounds are shown below.

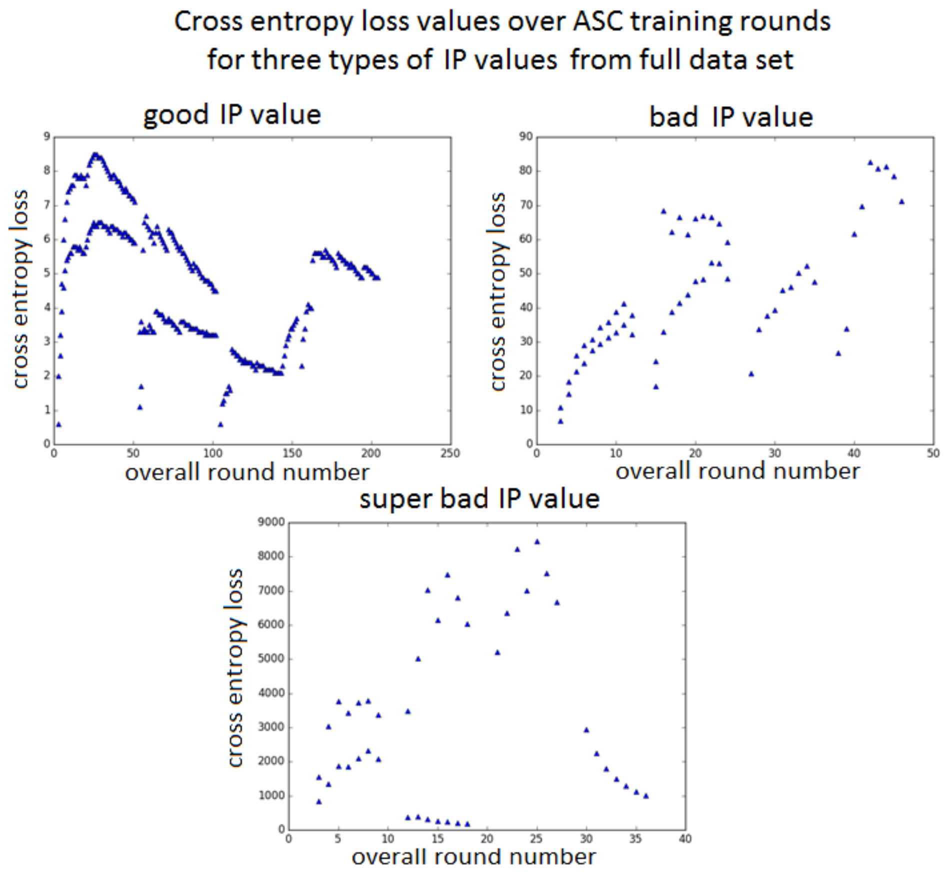

**Figure 19:** Cross entropy loss values for three types of IP values from the full data set.

In the full data set, we see jumps in cross entropy loss values as the overall round number increases (Figure 19). This is because when collecting data, we ran ASC on each IP value and program with multiple inputs. Each time ASC starts running a program and IP value with a new input, the initial cross entropy loss values start low. This phenomenon occurs for two reasons: 1) Early in a program's execution, very few bits have changed yet in the state of the program. 2) If ASC does not observe any previous change in a given bit in the state, then ASC will predict that the given bit will still stay the same in the future. Putting these to factors together, we find that early in a program's execution, before any bits have changed, ASC initially predicts that the state stays exactly the same in the future. Once a few bits begin to change, ASC's cross entropy loss begins to increase, but still does not get very high - even if ASC's future state prediction gets all of these bits wrong, these wrong bits are still relatively few in number. As the program continues to run, the program state changes more and ASC makes bolder predictions, resulting in a greater divergence in cross entropy loss values. Eventually, after this initial ramp up, ASC begins to to learn the actual underlying pattern of the states over time, and the cross entropy loss value may begin to decrease. This eventual decrease is what we are trying to predict. Overall, in the full data set figure, each "jump" to a lower cross entropy loss value marks each time we began running ASC with a new input.

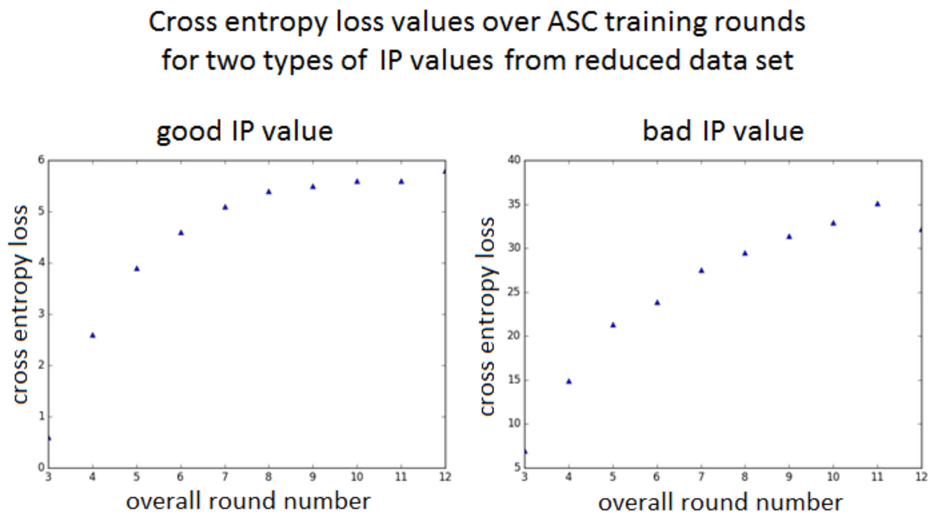

**Figure 20:** Cross entropy loss values for two types of IP values from the reduced data set.

While the full data set contains noise, each type of IP value still remains in a general neighborhood of cross entropy loss values. The good IP value's cross entropy loss values stay less than 10, the bad IP value's seem to approach 100, and the super bad IP value's loss values exceed 8000. The same is true for the reduced data set. Even though we keep only the first 10 rounds of ASC training on each IP value, we still see that within the first 10 rounds, the cross entropy loss values quickly diverge for the good and bad IP values (Figure 20). Thus, we don't lose the differentiation between the good and bad IP values in the reduced data set by taking only the first 10 rounds of ASC training for each IP value.

In the upcoming analysis of each machine learning model, we will present the results of testing each model on these three selected IP values. This way, not only can we compare the performance on each IP value within each machine learning model, but we can also compare the machine learning models with each other.

## 5.3 Bayesian ridge regression

We ran Bayesian ridge regression on both the full data set and the reduced data set. For each data set, we analyzed the performance of the Bayesian ridge regression with varying values of the shrinkage parameter $\lambda$. As described in Section 4, the shrinkage parameter $\lambda$ controls the extent to which we regulate overfitting. A higher $\lambda$ value means we overfit less to outliers in the data. We set $\lambda$ as each power of 10 between 1 and one million.

### 5.3.1 Selection of $\lambda$

For each value of $\lambda$, we computed both the average RMSE over 10 folds (10-fold RMSE) and the average coefficient of determination value over 10 folds (10-fold $r^2$). Recall from Section 5.2 that both the RMSE and $r^2$ values describe the overall prediction accuracy of the model. A low RMSE generally indicates a good fit, and an $r^2$ value above 0 and closer to 1 also indicates a good fit. We computed these for both the full data set and the reduced data set.

As shown in Figures 21 and 22, for both the full and reduced data sets, the values of $\lambda$ that produced both the lowest RMSE values and the highest $r^2$ values were $\lambda < 100$. Given the large size of the data set, an optimal $\lambda < 100$ is relatively small. Higher values of $\lambda$ would have helped the model perform better if the data set contained a greater number of outliers and was highly prone to overfitting. This suggests that both the full and reduced data sets are not actually heavily affected by overfitting by Bayesian ridge regression.

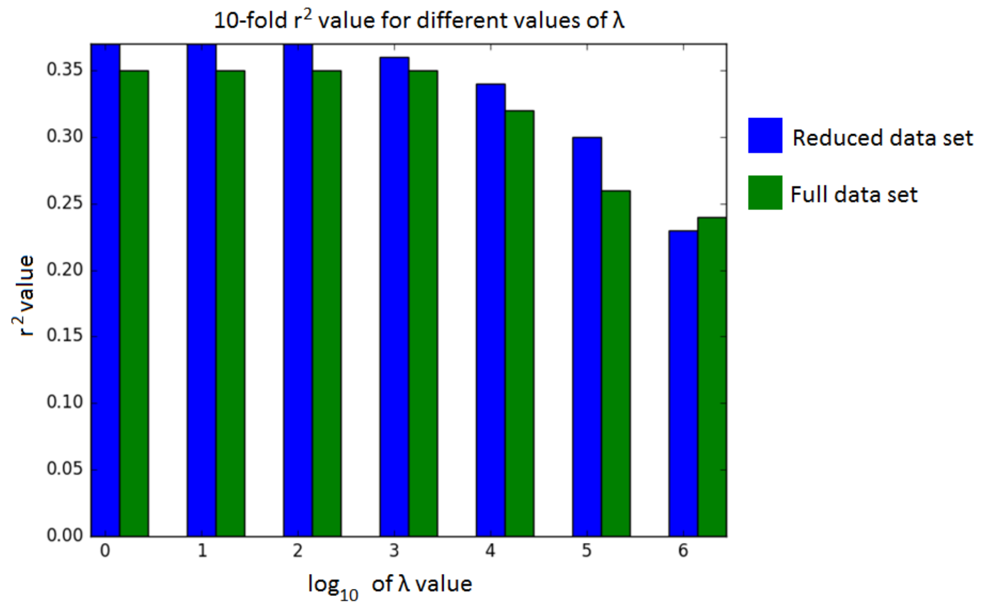

**Figure 21:** Plot of 10-fold $r^2$ values for different values of $\lambda$ for Bayesian ridge regression. The $r^2$ values for the reduced data set are plotted as the blue bars on the left, and the $r^2$ values for the full data set are plotted as the green bars on the right. For both data sets, the highest $r^2$ values occur for $\lambda$ between 1 and 100. The change in $r^2$ value with respect to $\lambda$ is slightly greater for the reduced data set.

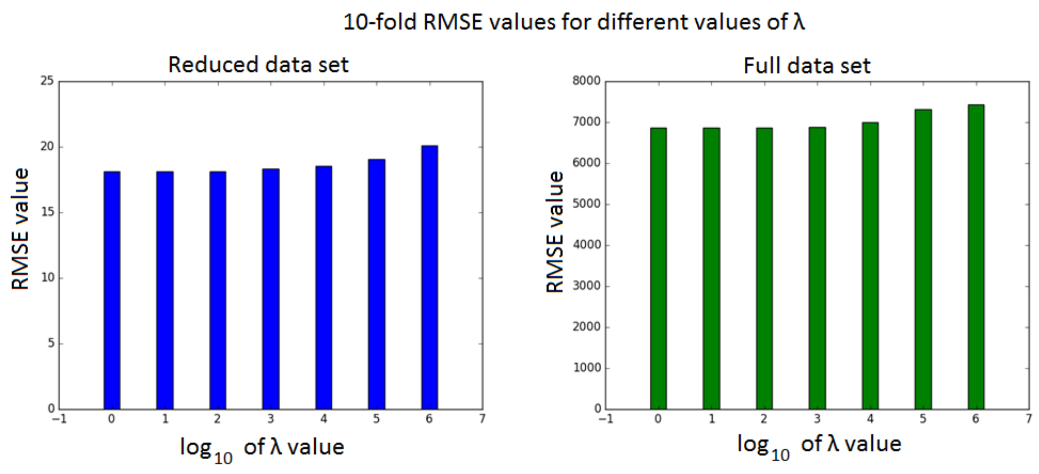

**Figure 22:** Plot of 10-fold RMSE values for different values of $\lambda$ for Bayesian ridge regression. The RMSE values for the reduced data set are plotted as the blue bars on the left, and the RMSE values for the full data set are plotted as the green bars on the right. For both data sets, the lowest RMSE values range between 1 and 100.

### 5.3.2 Overall prediction accuracies

The value of $\lambda$ that performed the best over both the full and reduced data sets was $\lambda = 10$. Statistically, $\lambda = 10$ did not perform significantly better than $\lambda = 1$ or $\lambda = 100$. Still, for simplicity, we will show results for only $\lambda = 10$ in our overall performance analysis.

Summary statistics for Bayesian ridge regression, $\lambda = 10$

| Data set | 10-fold RMSE | 10-fold $r^2$ |
|---|---|---|
| Full data set | 6864.32 | 0.35 |
| Reduced data set | 18.11 | 0.37 |

**Figure 23:** Summary statistics for Bayesian ridge regression run on both the full data set and the reduced data set. These are for $\lambda = 10$, which produced the best results for both the full and reduced data sets.

For a more detailed evaluation of the performance of Bayesian ridge regression, we show the $r^2$ and RMSE values in more detail for $\lambda = 10$. The $r^2$ value is greater than 0, which indicates that the Bayesian ridge regression produced a better fit than the most naive method of always choosing the sample mean of the target values. However, the $r^2$ value is still far from 1. This is expected, since we expect that the relationship between the input features and the cross entropy loss is likely non-linear and more complicated than anything a single linear regression can model. It is also unsurprising that the RMSE value for the full data set is much higher than the RMSE value for the reduced data set. We expect this because the full data set contains many more data points with a much higher variance. We present a more detailed graphical representation of the prediction accuracies for each data set below.

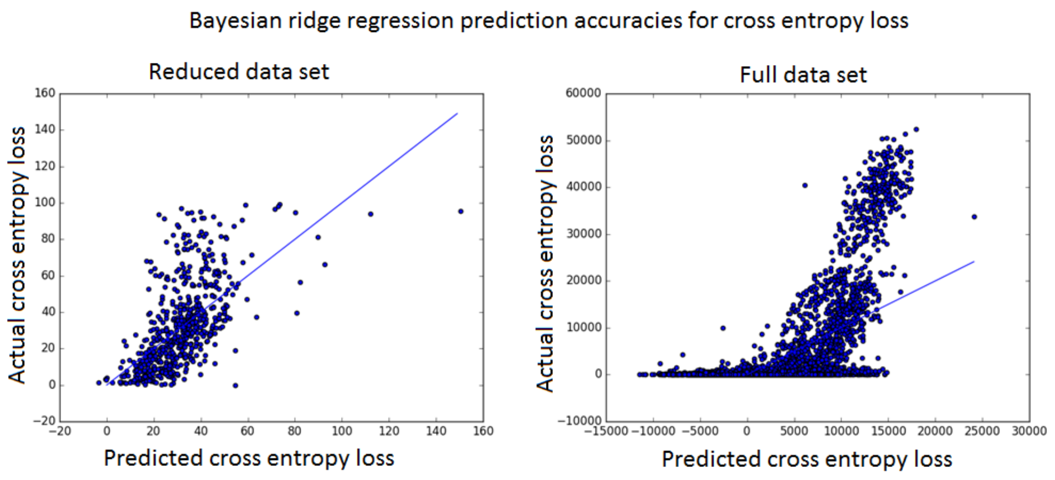

**Figure 24:** Plot of prediction accuracies on a randomly selected test set for Bayesian ridge regression with $\lambda = 10$. The predicted cross entropy loss is plotted on the x axis, and the actual cross entropy loss is plotted on the y axis. The overlayed line represents the line for which actual = predicted. The plot on the left shows prediction accuracies for the reduced data set, and the plot on the right shows prediction accuracies for the full data set.

In the scatter plots of prediction accuracies (Figure 24), the predictions for the reduced data set seem to adhere much more closely to the actual = predicted line. On the other hand, for the full data set, we appear to underestimate the predicted cross entropy loss for actual cross entropy loss values greater than 20000. This result is expected, since in the preliminary data analysis, we observed that the actual cross entropy loss values are highly skewed right, and values greater than 20000 form the tail of the data set. Thus, if controlling for overfitting, the predicted cross entropy loss values would also be skewed towards smaller values.

Another important observation for the full data set is that the predicted cross entropy loss values can actually be negative for cross entropy loss values close to 0. This suggests that in trying to fit the majority of the data, the Bayesian ridge regression also loses accuracy with the smallest target values. This is not desirable in practice, since the greatest variability in ASC speedup in practice actually occurs for cross entropy loss values less than 100. As expected, training on the reduced data set produces a much better fit for data points with smaller target values and would as a result be much more useful for ASC IP selection in practice.

### 5.3.3 Interpretation of feature weights

In addition to performance analysis, we can also interpret the Bayesian ridge regression model to understand which features it considers most "important" when predicting the target values. We do this by analyzing the feature weights. Recall that in a linear regression model, we directly compute the target values using the weight vector $\hat{\mathbf{w}}$:

$$\hat{y_i} = \hat{\mathbf{w}}^\top \mathbf{x}_i$$

This means that for each feature, the corresponding weight tells us the correlation between that feature value and the target value. Weight values of 0 indicate no correlation between the feature and the predicted target value. Negative weight values indicate a negative correlation between the feature and the predicted target value, and positive weight values indicate a positive correlation.

The performance analysis of the Bayesian ridge regression suggests that the model produces the best fit and most meaningful results when run on the reduced data set. Thus, we analyze the weight vector output by the model when run on the reduced data set.

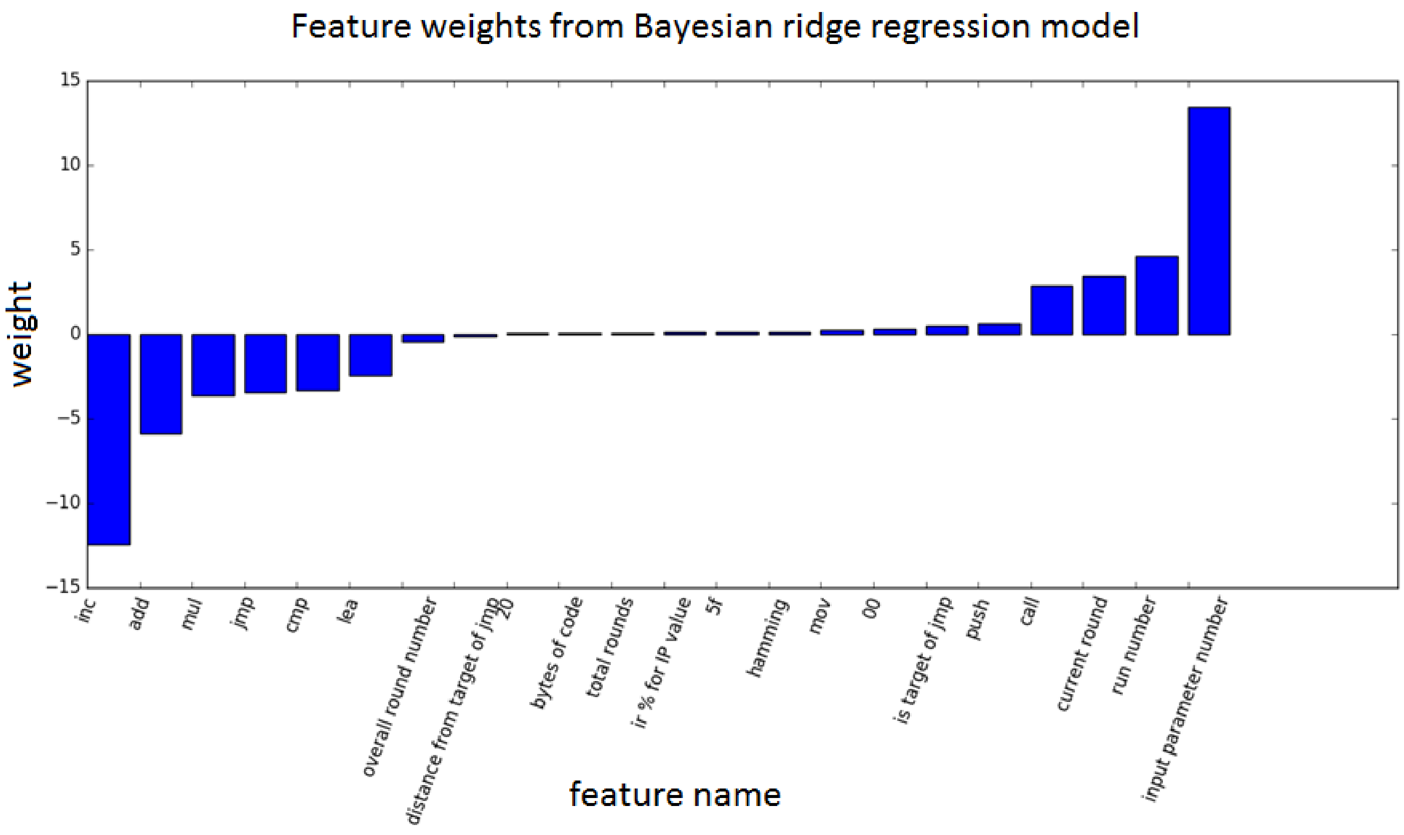

**Figure 25:** Feature weights for the weight vector output by Bayesian ridge regression with $\lambda = 10$ run on the reduced data set. Features with weights less than 0.05 are not displayed. The feature names are abbreviated as follows: 1. hexdump 1-gram counts are labeled as the 1-gram itself (so the feature "00" denotes the count of the 1-gram "00" in the program binary). 2. The binary feature for the instruction type of the IP value is labeled as the name of the instruction type itself (so the feature "inc" denotes a binary feature that is equal to 1 if the IP value references an "inc" instruction, and 0 otherwise).

As expected, the features with the highest weight values (and thus the highest positive correlation with the predicted target value) are the current round, the run number, and the input parameter number. Together, these three features simply describe how long ASC has been training on a given program and IP value. In general, the Bayesian linear regression predicts that cross entropy loss increases with more ASC training. This is generally true for most data points, since most programs and IP values are "bad". Only a few programs and IP values actually show decreasing cross entropy loss values over ASC training rounds.

Another interesting takeaway from the analysis of the feature weights is that the Bayesian ridge regression predicts that the cross entropy loss should be lower if the IP value references an inc instruction. We do not have a simple explanation for this, but it is an interesting point for future investigation. Perhaps the predictability of an IP value is somehow related to the instruction that the IP value refers to.

Aside from displaying the "important" features, this figure also reveals which features do not correlate with the predicted target values. Most of the hexdump features are not displayed in the figure because their weights are too close to 0. In fact, most of the features that are displayed appear to be specific to the IP value or the round number, and not to the program. Features associated with the individual programs, such as cache usage and call graph features, were given very little weight by the Bayesian linear regression.

### 5.3.4 Performance on individual IP values

In addition to evaluating the overall performance of the Bayesian ridge regression, we also evaluate its performance on individual good, bad, and super bad IP values.

For each IP value, we vary the way we construct the training sets. In all previous evaluation, the training and test sets were split at random without consideration for which programs the data points came from. Thus, the test set and the training set could both contain data points from the same program. However, we would ultimately like to be able to make predictions for a given program and IP value without seeing any prior training data from that program. Thus, in this section, we evaluate the model's performance on each type of IP value with three different types of training sets:

1. Full training set: the training set consists of the entire data set, including data points from the same program as the IP value being tested.
2. Program only: the training set consists only of data points from the same program as the IP value being tested. This means that the training set still contains all of the different IP values from that program, but we make predictions on only one of the IP

values from that program. There is still a great deal of variation of cross entropy loss across the different IP values within a program.

3. All except the program: the training set consists of the entire data set except for data points from the same program as the IP value being tested.

We first analyze the performance on the reduced data set, since the data is less noisy and thus easier to interpret than the full data set. Our results show that the prediction accuracy varies greatly between the good and bad IP values and programs. The Bayesian ridge regression appears to make the best predictions for the bad IP value when trained on the full training set (Figure 27). It appears to make far worse predictions for the good IP value, not actually capturing the low cross entropy loss values (Figure 26). This means that the Bayesian ridge regression is heavily influenced by the bad IP values relative to the good IP values. This is unsurprising, since there are far more bad IP values than good IP values in the data set.

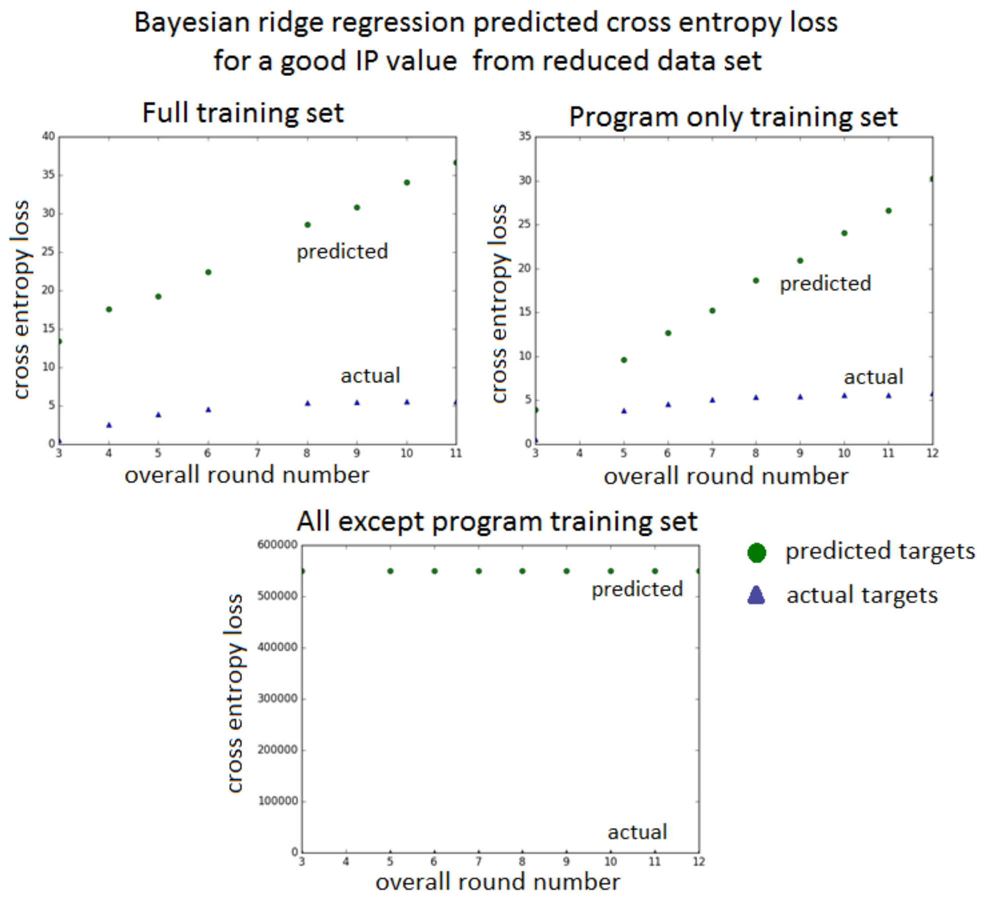

**Figure 26:** Predicted and actual cross entropy loss values for the good IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 22.07$, $r^2 = -169.91$; Program only: $RMSE = 14.92$, $r^2 = -91.95$; All except program: $RMSE = 549666.24$, $r^2 = -126031369202.0$

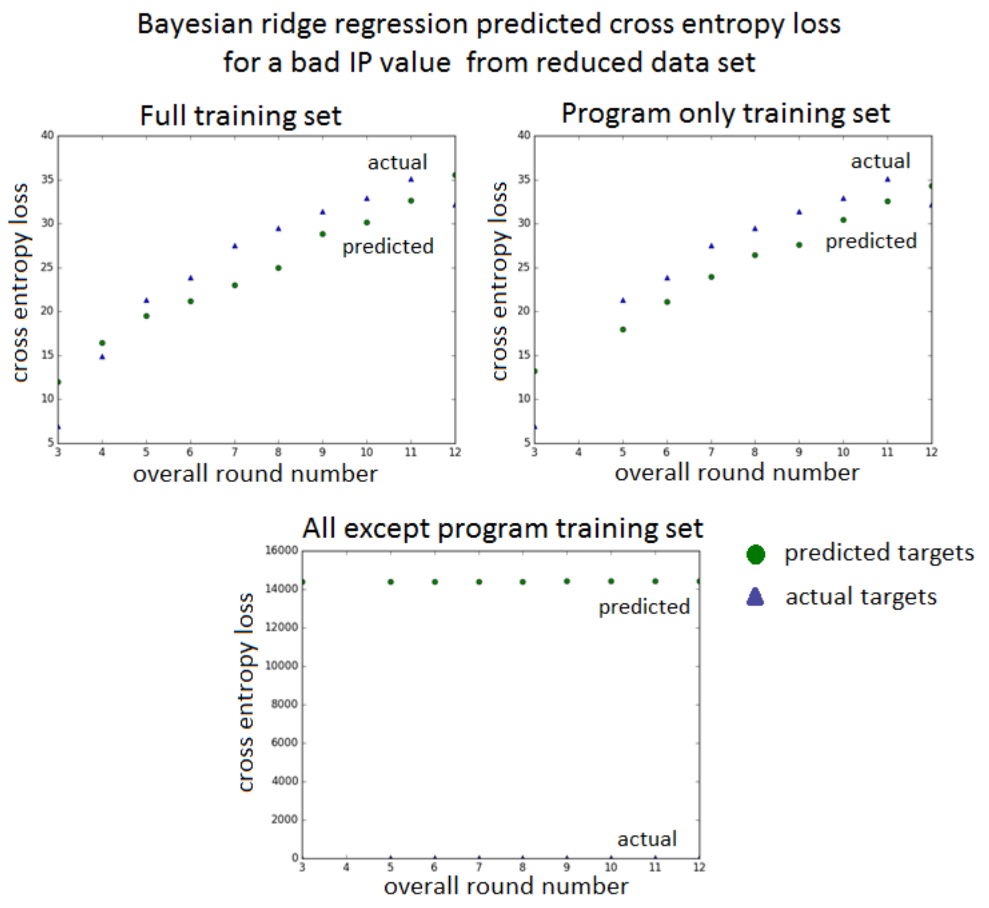

**Figure 27:** Predicted and actual cross entropy loss values for the bad IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 3.31$, $r^2 = 0.84$; Program only: $RMSE = 3.49$, $r^2 = 0.81$; All except program: $RMSE = 14385.90$, $r^2 = -3131019.47$

Similar to the reduced data set results, the full data set results also show that the prediction accuracy for the cross entropy loss values varies greatly depending on the type of IP value (Figures 28, 29, and 30). The Bayesian ridge regression appears to make the closest predictions for the super bad IP value when trained on the full training set (Figure 30). It appears to make the worst predictions for the good IP value (Figure 28). This means that when given the full data set, the Bayesian ridge regression is heavily influenced by the bad and super bad IP values relative to the good IP values.

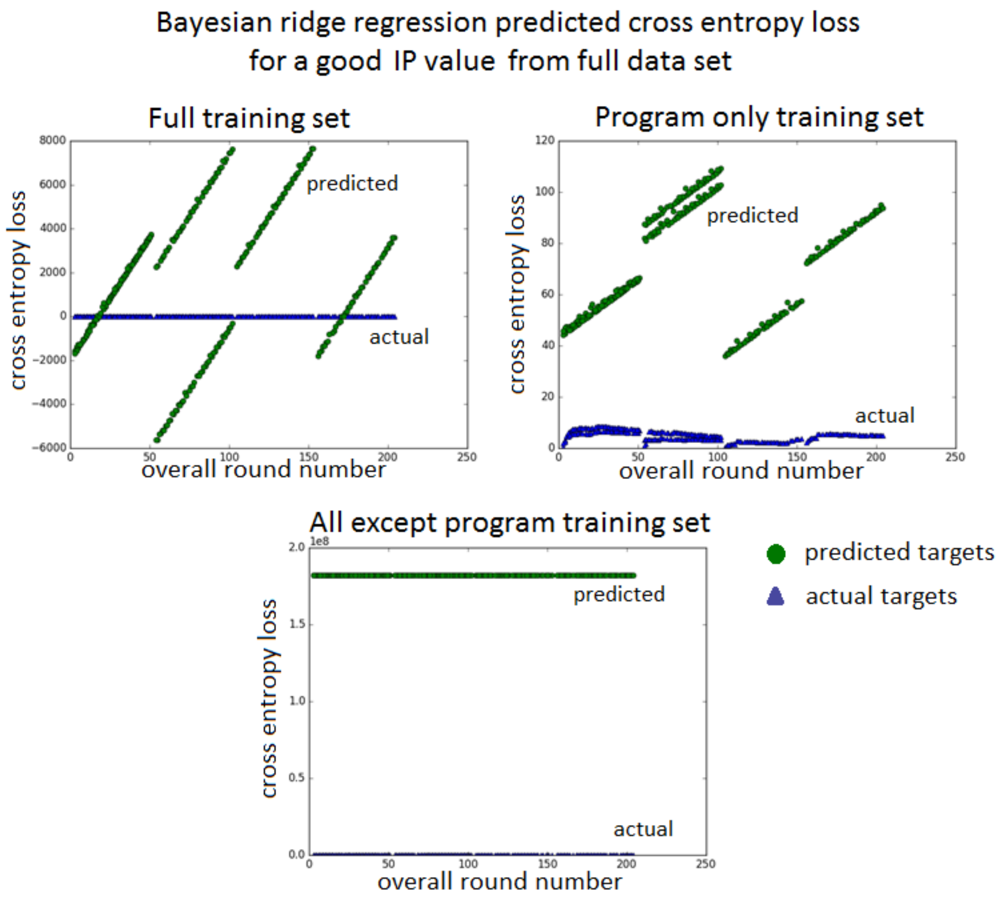

**Figure 28:** Predicted and actual cross entropy loss values for the good IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 3603.06$, $r^2 = -3710882.26$; Program only: $RMSE = 70.69$, $r^2 = -1405.58$; All except program: $RMSE = 182271471.77$, $r^2 = -9.34 \times 10^15$

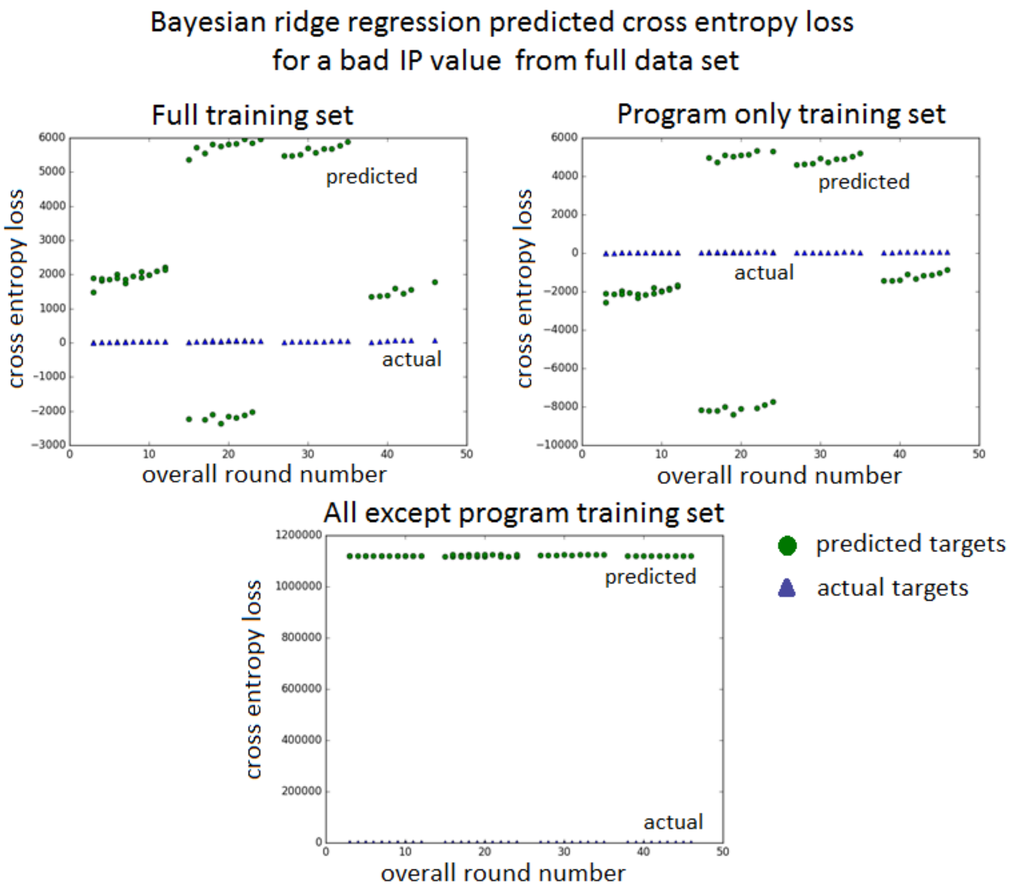

**Figure 29:** Predicted and actual cross entropy loss values for the bad IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 3772.20$, $r^2 = -44429.53$; Program only: $RMSE = 4588.33$, $r^2 = -58715.52$; All except program: $RMSE = 1120811.73$, $r^2 = -3503608987.33$

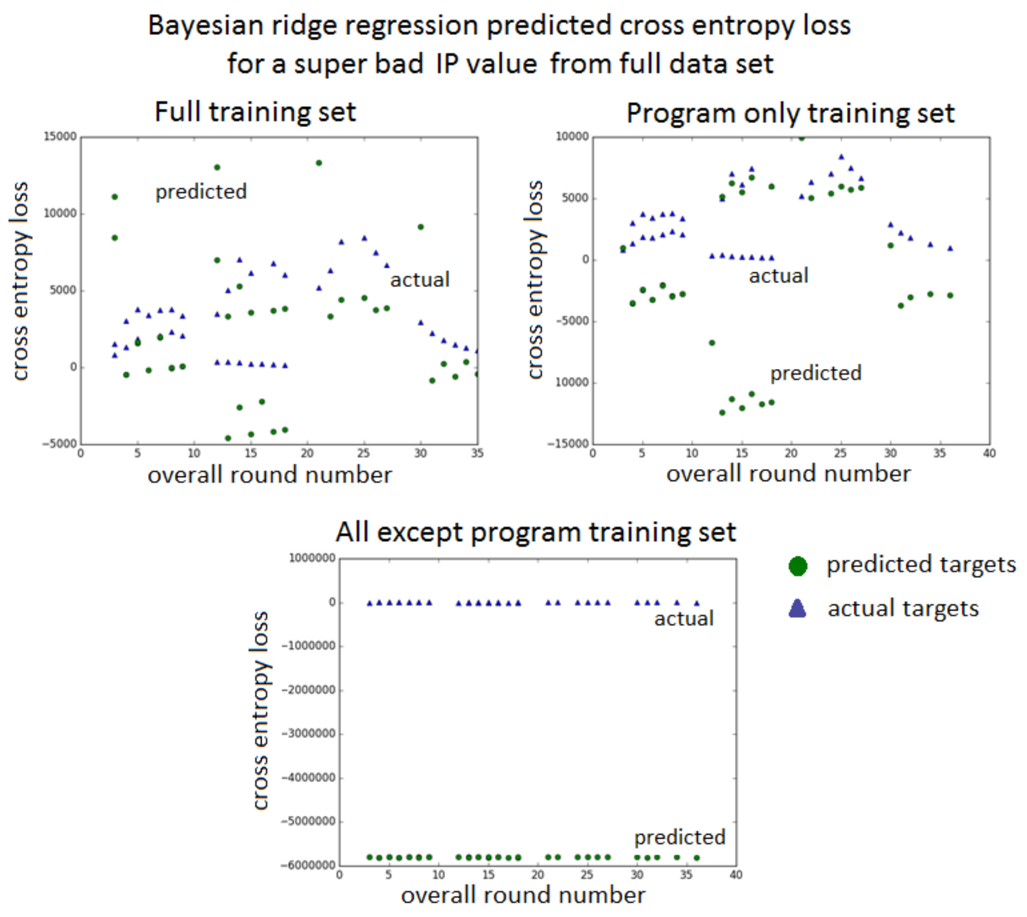

**Figure 30:** Predicted and actual cross entropy loss values for the super bad IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 13668.88$, $r^2 = 0.21$; Program only: $RMSE = 10398.60$, $r^2 = 0.55$; All except program: $RMSE = 5814840.65$, $r^2 = -140263.43$

Breaking down the different training sets for each type of IP value, we observe that the performance when trained on the full training set is similar to the performance when trained on data points only from the same program as the IP value being tested. On the other hand, when trained on all data points except those from the same program as the IP value being tested, the predictions were dramatically worse. This was true for good, bad, and super bad IP values. This implies that currently, the Bayesian ridge regression relies heavily on training data from the program of the IP value being tested to make predictions for that IP value. Unfortunately, this means that the Bayesian ridge regression does not achieve the holy grail goal of being able to make predictions for the IP values of a program without ever having seen that program.

Given the analysis of the feature weights in the previous section, perhaps it is not entirely surprising that the Bayesian ridge regression performs poorly when the data points from the same program as the IP value being tested are removed from the training set. The Bayesian ridge regression placed almost no weight on any features specific to the program. This means that when the data points from the same program as the IP value being tested were removed, the statistical strength of all of those data points was also removed. The model then had no way to differentiate the tested IP value from all of the other data points belonging to other programs, and thus made poorer predictions for that IP value.

Still, this does not mean that the model does not have the potential to be useful. It is still useful in practice to be able to predict the performance of a single IP value given data from the other IP values of a program. For instance, if a user runs ASC on a single program for a few IP values, the user can then use the data from those runs to predict ASC's cross entropy loss values for other IP values for that program as well.

## 5.4 Gaussian process model

To see if we could improve from the Bayesian ridge regression model, we also ran the Gaussian process model and analyzed its performance on the reduced data set. Due to hardware and time constraints, we were not able to achieve convergence for the Gaussian process model on the full data set. Thus, in this project, we will not be able to evaluate the Gaussian process model's ability to predict super bad program and IP values. However, as far as ASC speedup is concerned, this is not a huge loss - as long as we can accurately categorize IP values as good or bad, then we can meaningfully predict whether programs and IP values will benefit from being run with ASC.

### 5.4.1 Overall prediction accuracies

Summary statistics for Gaussian process

| Data set | 10-fold RMSE | 10-fold $r^2$ |
|---|---|---|
| Reduced data set | 20.39 | 0.12 |

**Figure 31:** Summary statistics for Gaussian process run on the reduced data set.

Because the Gaussian process should theoretically be able to model more complicated, non-linear underlying functions from feature vectors to cross entropy loss values, we expected the Gaussian process model to perform better overall than the Bayesian ridge regression. However, the 10-fold RMSE and the 10-fold $r^2$ values both turned out to be worse than those of the Bayesian linear regression. Perhaps if we allowed the Gaussian process to train for longer amounts of time with different regularization settings, the gap in performance would close. The reasons for this performance gap would be interesting to explore in future work.

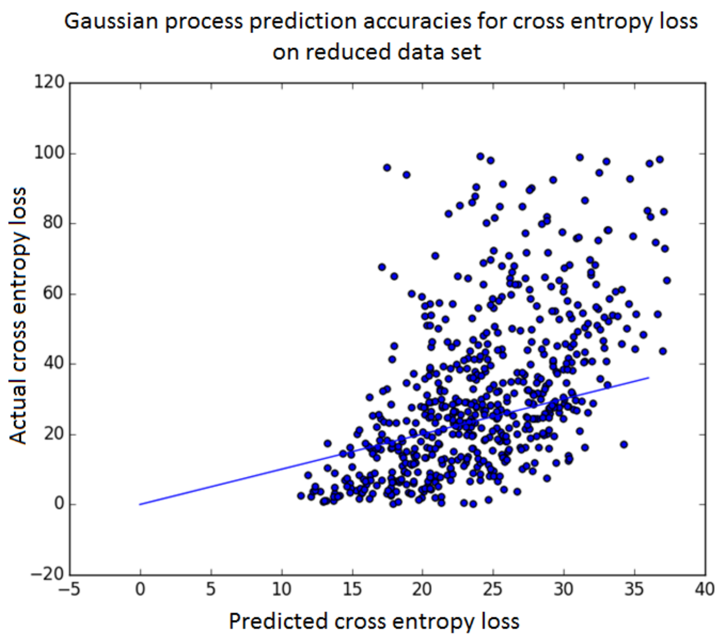

**Figure 32:** Plot of prediction accuracies on a randomly selected test set for the Gaussian process model. The predicted cross entropy loss is plotted on the x axis, and the actual cross entropy loss is plotted on the y axis. The overlayed line represents the line for which actual = predicted.

Based on the plot of prediction accuracies (Figure 32), the Gaussian process model appears to underestimate many high cross entropy loss values. This phenomenon occurs in the Bayesian ridge regression as well; thus, it does not come as a surprise, and does not show any particular difference in performance between the models.

### 5.4.2 Performance on individual IP values

Just as we did for the Bayesian ridge regression, we also evaluate the performance of the Gaussian process on good and bad IP values. We do this for each of three different types of training sets: a full training set, a training set consisting only of data points from the same program as the IP value being tested, and a training set consisting of all data points except for those from the same program as the IP value being tested.

One interesting result from running the Gaussian process model on different training sets is that the Gaussian process model did not exhibit any dramatic decrease in performance when the test program's data was not included in the training set (Figures 33 and 34). This was true for both the good IP value and the bad IP value. This shows that even though the Gaussian process model achieved a lower $r^2$ value than the Bayesian ridge regression when trained on the full data set, the Gaussian process model was robust enough to work just as well when the data from the program of the IP value being tested was removed from the training set.

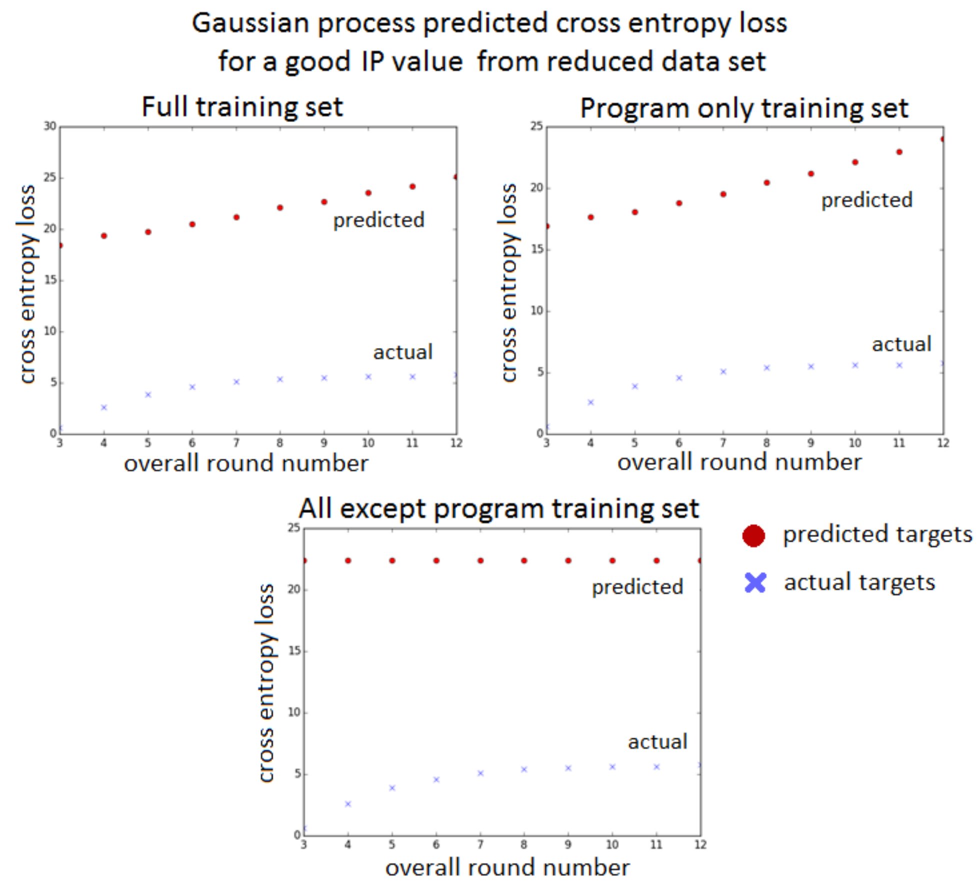

**Figure 33:** Gaussian process predicted and actual cross entropy loss values for the good IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 17.24$, $r^2 = -115.84$; Program only: $RMSE = 15.76$, $r^2 = -96.57$; All except program: $RMSE = 17.98$, $r^2 = -126.07$

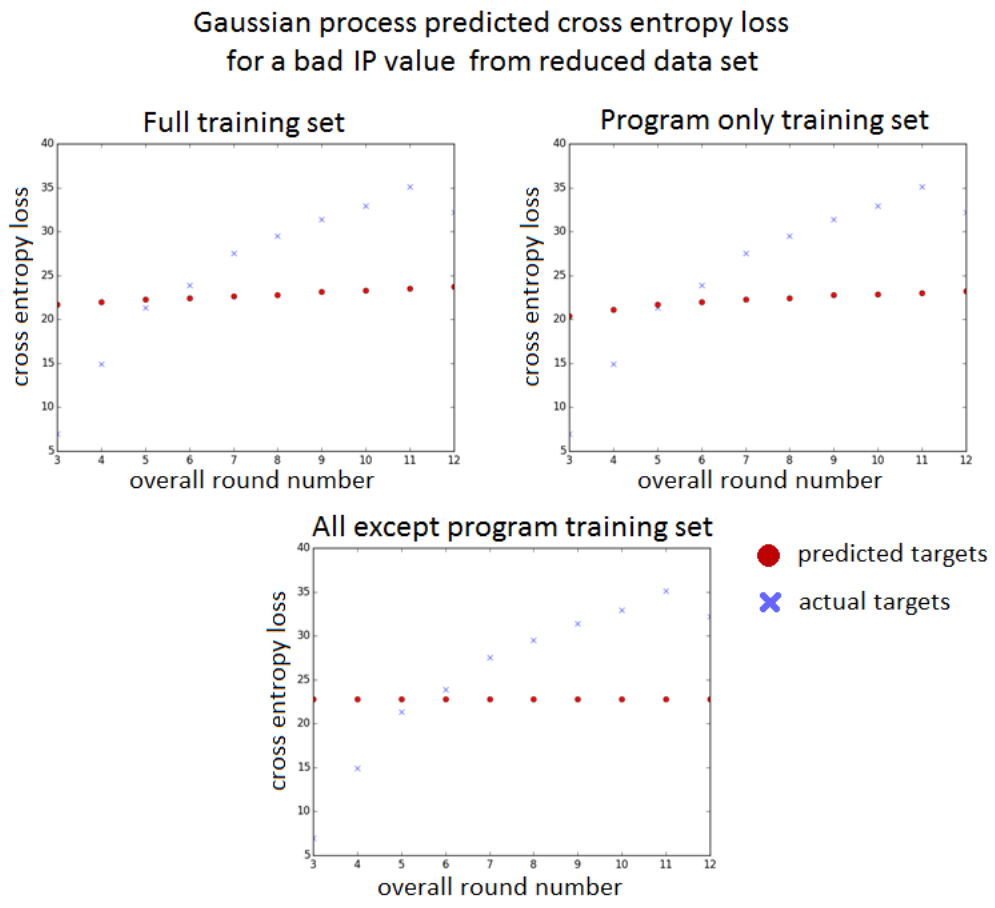

**Figure 34:** Gaussian process predicted and actual cross entropy loss values for the bad IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 8.38$, $r^2 = 0.025$; Program only: $RMSE = 8.38$, $r^2 = 0.02$; All except program: $RMSE = 8.94$, $r^2 = -0.10$

An important implication of the Gaussian process model's robustness is that the feature set is good enough to show some relationship between a program and ASC's cross entropy loss values. If the Gaussian process can make predictions for a program that it has never seen before, then there must actually be some correlation between the program features and ASC's cross entropy loss values.

Perhaps one reason for the robustness of the Gaussian process model is that it can capture non-linear relationships. Unlike the Bayesian ridge regression, the Gaussian process model allows us to model an arbitrary non-linear function that maps feature vectors $\mathbf{x}_i$ to target cross entropy loss values. Perhaps there is a more complicated non-linear relationship between the program specific features, the IP value specific features, and the cross entropy loss values, and the Gaussian process model is able to capture this better than the Bayesian ridge regression.

Even though the actual performance of the Gaussian process model is still not good enough to be used in practice, we can still be optimistic in saying that the feature set collected in this project is good enough to warrant further exploration with different models.

## 5.5 Multi-task Lasso regression

While the two single-task regression models showed promising results, we also wanted to see if we could get even better results with a multi-task regression model. We were able to run multi-task Lasso regression on both the full data set and the reduced data set. As we did for Bayesian ridge regression, we also ran multi-task Lasso regression with different values of $\lambda$. We set $\lambda$ as each power of 10 between 1 and $10^{16}$.

### 5.5.1 Selection of $\lambda$

For each value of $\lambda$, we computed both the 10-fold RMSE and the 10-fold $r^2$ value. We plot the results in the following figures.

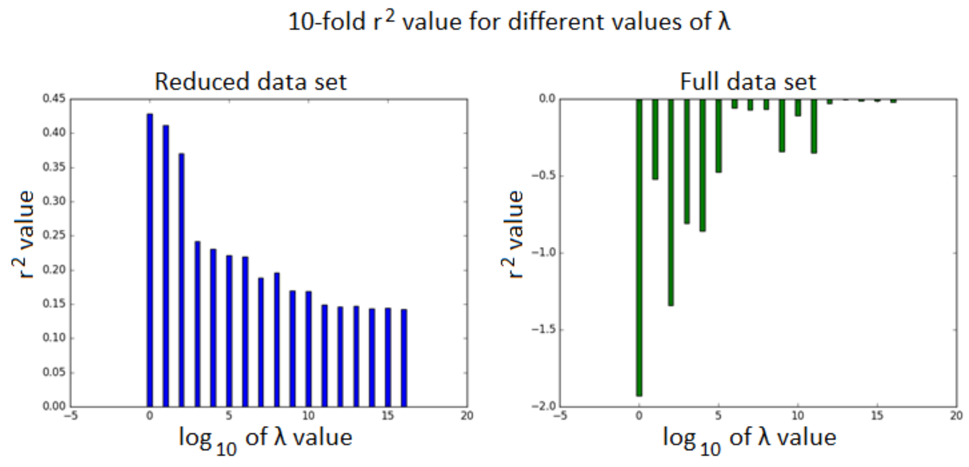

**Figure 35:** Plot of 10-fold $r^2$ values for different values of $\lambda$ for Multi-task Lasso regression. The $r^2$ values for the reduced data set are plotted as the blue bars on the left, and the $r^2$ values for the full data set are plotted as the green bars on the right.

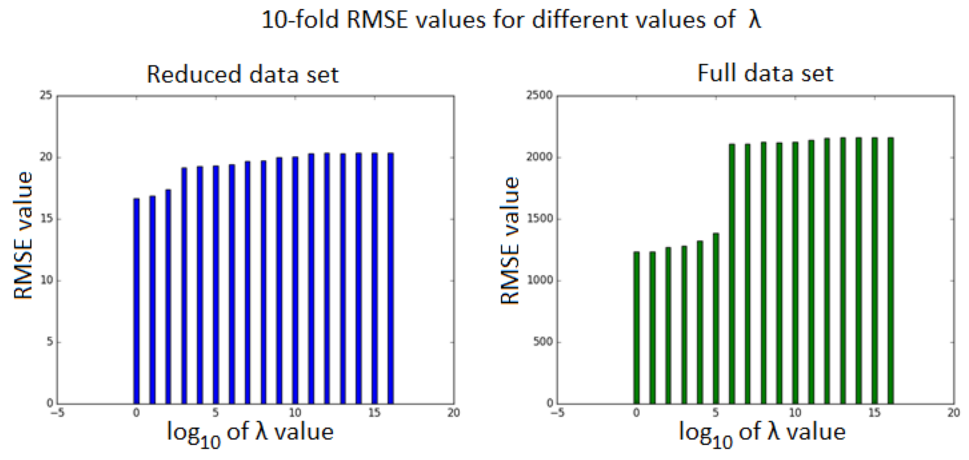

**Figure 36:** Plot of 10-fold RMSE values for different values of $\lambda$ for Bayesian ridge regression. The RMSE values for the reduced data set are plotted as the blue bars on the left, and the RMSE values for the full data set are plotted as the green bars on the right.

Unlike the Bayesian ridge regression, the multi-task Lasso regression showed different optimal $\lambda$ values for the reduced data set and full data set (Figures 35 and 36). For the reduced data set, the highest $r^2$ values and lowest RMSE values occurred at $\lambda < 100$. For the full data set, the highest $r^2$ values occurred at $\lambda > 10^{12}$. This suggests that the full data set receives more performance penalty from overfitting by multi-task Lasso regression than the reduced data set. This is expected, since the full data set has a wider range of target values with a higher number of large outliers than the reduced data set.

This discrepancy between the full and reduced data sets may seem contradictory when we recall that the Bayesian ridge regression did not see high performance improvements for high values of $\lambda$ for the full data set. However, it is important to note that the Bayesian ridge regression and multi-task Lasso regression are fundamentally different models, and thus will have different tendencies for overfitting. Also, the multi-task regression only takes into account the first 10 data points for each IP value of a program. For many of the super bad IP values, the full data set contains much more than 10 rounds per IP value. This is because the super bad IP values were often the type to be encountered at a high frequency without any predictable pattern in the program state. When the multi-task regression effectively removes most of these rounds from the data set, it decreases the frequency of data points from the super bad IP values. This may make them act more as outliers when training the model.

### 5.5.2 Overall prediction accuracies

The value of $\lambda$ that performed the best for the full data set was $\lambda = 10^{13}$, and the value of $\lambda$ that performed the best for the reduced data set was $\lambda = 1$. For subsequent performance analysis, we will show results for only these two optimal $\lambda$ values.

Summary statistics for multi-task Lasso regression

| Data set | 10-fold RMSE | 10-fold $r^2$ |
|---|---|---|
| Full data set ($\lambda = 10^{13}$) | 2161.07 | -0.012 |
| Reduced data set ($\lambda = 1$) | 16.65 | 0.43 |

**Figure 37:** Summary statistics for multi-task Lasso regression run on both the full data set and the reduced data set. The 10-fold RMSE and $r^2$ values are shown for the best values of $\lambda$ for each data set.

Analyzing the $r^2$ and RMSE values in more detail for the optimal values of $\lambda$, we find that the multi-task Lasso regression actually performed the best on the reduced data set out of all of the machine learning methods presented so far. The multi-task Lasso regression produced both the highest 10-fold RMSE and the lowest $r^2$ value on the reduced data set. Perhaps the sharing of statistical strength across a given IP value and program plays an important role when modeling the reduced data set.

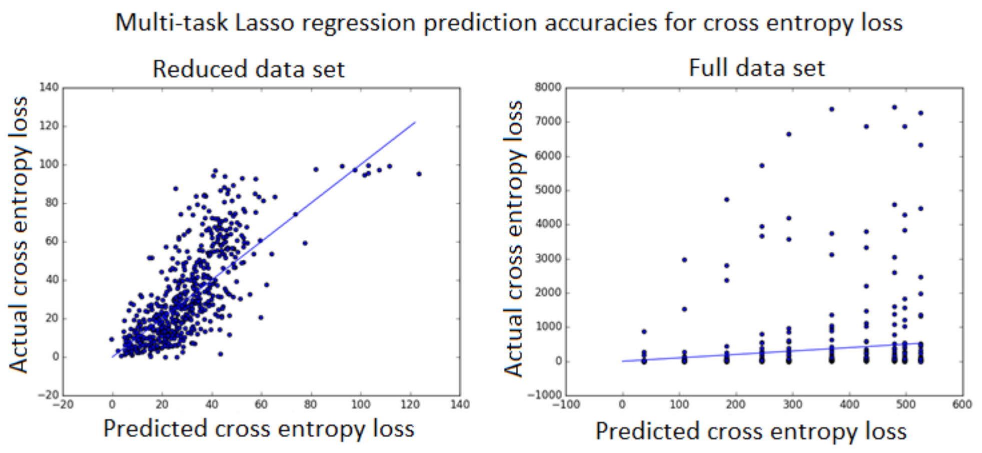

**Figure 38:** Plot of prediction accuracies on a randomly selected test set for the multi-task Lasso regresion. The predicted cross entropy loss is plotted on the x axis, and the actual cross entropy loss is plotted on the y axis. The overlayed line represents the line for which actual = predicted.

In the scatter plots of actual and predicted cross entropy loss for the multi-task Lasso regression (Figure 38), we observe that the multi-task Lasso regression again appears to yield similar performance to the Bayesian ridge regression on the reduced data set.

For the full data set, on the other hand, the multi-task Lasso regression performed worse overall than the Bayesian ridge regression. We observe this in both the scatter plots and the $r^2$ values (Note that we cannot actually compare the RMSE values between the Bayesian ridge regression and the multi-task Lasso regression, since the multi-task Lasso regression is limited to only the first 10 rounds for each IP value and program). For the full data set, it turns out that the Lasso regression achieved the best performance by predicting the exact same 10 cross entropy loss values for every IP value and program. This uniformity is due to the high optimal $\lambda$ value that reduces overfitting. This is actually completely not useful in practice, since it does not differentiate between good and bad IP values at all. Thus, for future analysis, it would be useful to explore different ways of better fitting the multi-task Lasso regression model to the full data set.

### 5.5.3 Performance on individual IP values

We also analyze the performance of the multi-task regression on individual IP values. First, we analyze the performance on the reduced data set, for which the multi-task Lasso regression achieved relatively good predictions.

Like the Bayesian ridge regression, the multi-task Lasso regression on the reduced data set also performed much worse when the test program data was removed from the training set (Figures 39 and 40). This suggests that like the Bayesian ridge regression, the multi-task Lasso regression also relies on the features of the IP value more than the features of the program.

When trained on the full training set, the multi-task Lasso regression also achieved a much closer fit to the bad IP value than the good IP value in the reduced data set (Figures 39 and 40). This could be attributed to the many more bad IP values than good IP values in the data set. If the multi-task Lasso regression fit too closely to the outlying good IP values, then it would perform worse overall.

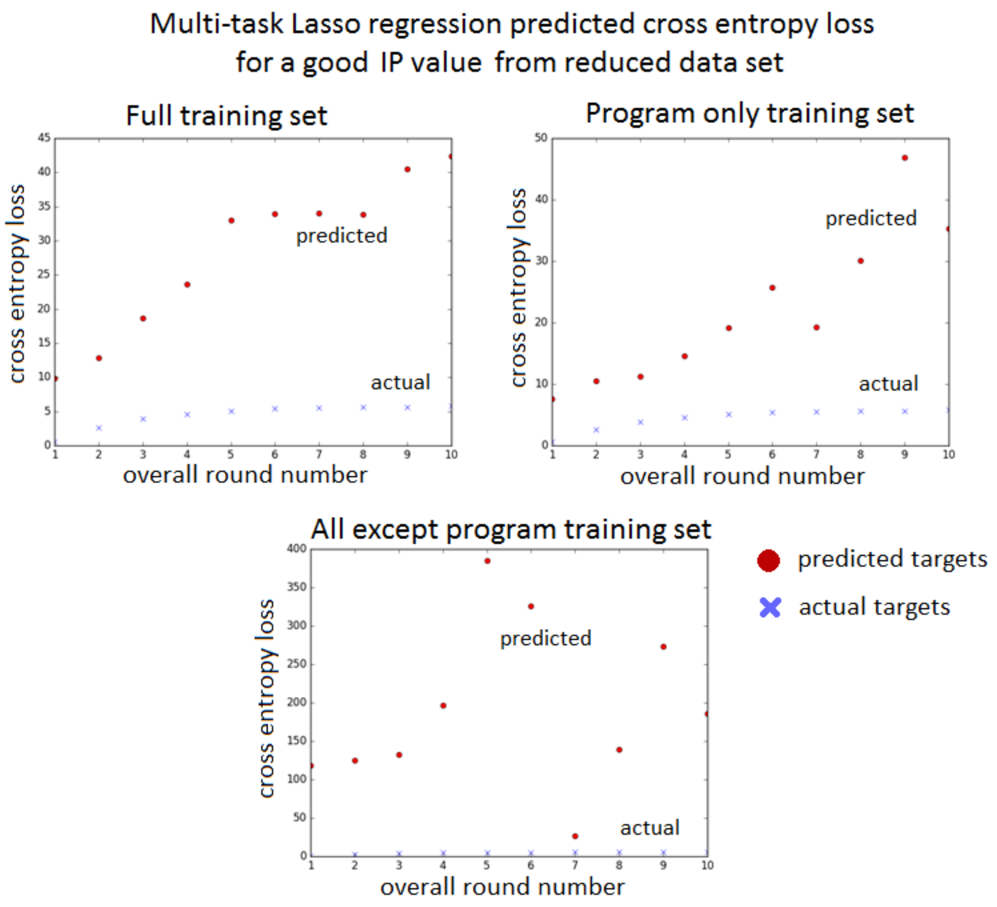

**Figure 39:** Predicted and actual cross entropy loss values for the good IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 25.54$, $r^2 = -255.38$; Program only: $RMSE = 20.58$, $r^2 = -165.34$; All except program: $RMSE = 212.48$, $r^2 = -17731.38$

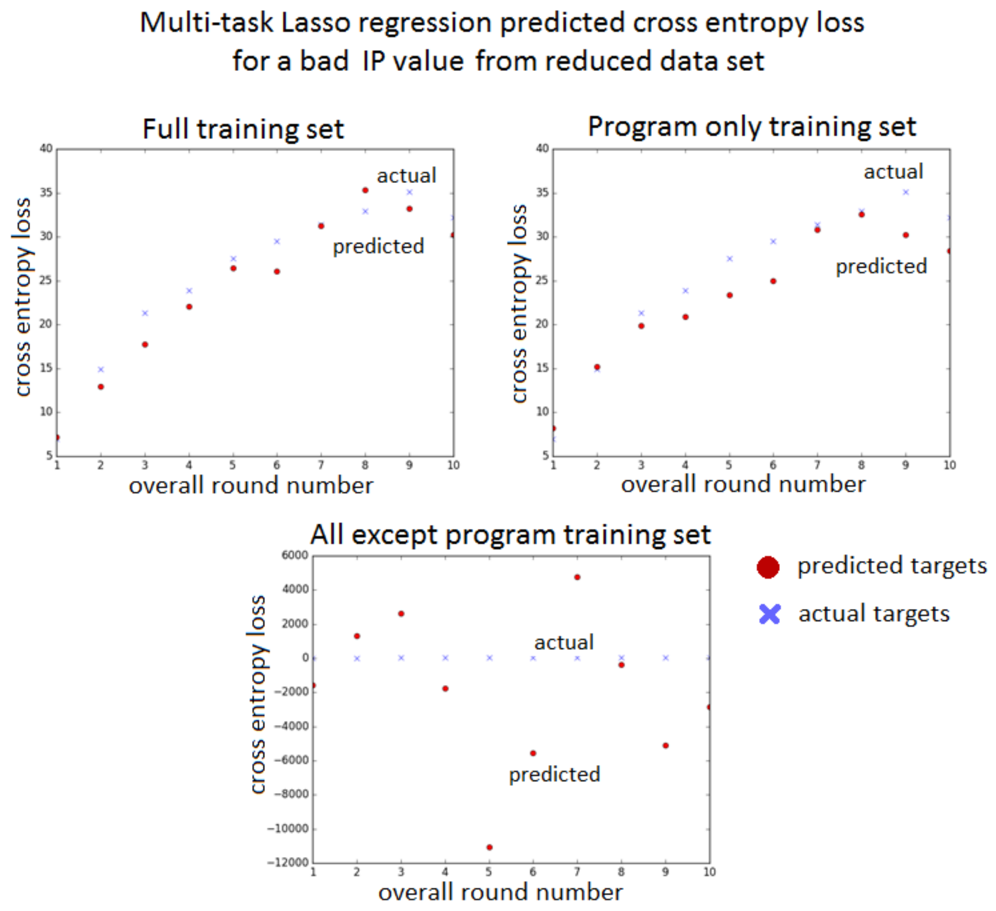

**Figure 40:** Predicted and actual cross entropy loss values for the bad IP value and program trained on three different types of training sets. This comes from the reduced data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 2.14$, $r^2 = 0.93$; Program only: $RMSE = 2.97$, $r^2 = 0.87$; All except program: $RMSE = 4741.65$, $r^2 = -311668.89$

Since the multi-task regression outputs the exact same 10 cross entropy loss values for each IP value in the full data set, it is not surprising that when the test program was removed from the training set, the performance of the model did not change very much (Figures 41, 42, and 43). This is because the overall range of cross entropy loss values for each program does not differ very much between programs in the full data set. Thus, the removal of one program from the training set had little effect on the multi-task regression's rough fit of 10 cross entropy loss values to each IP value.

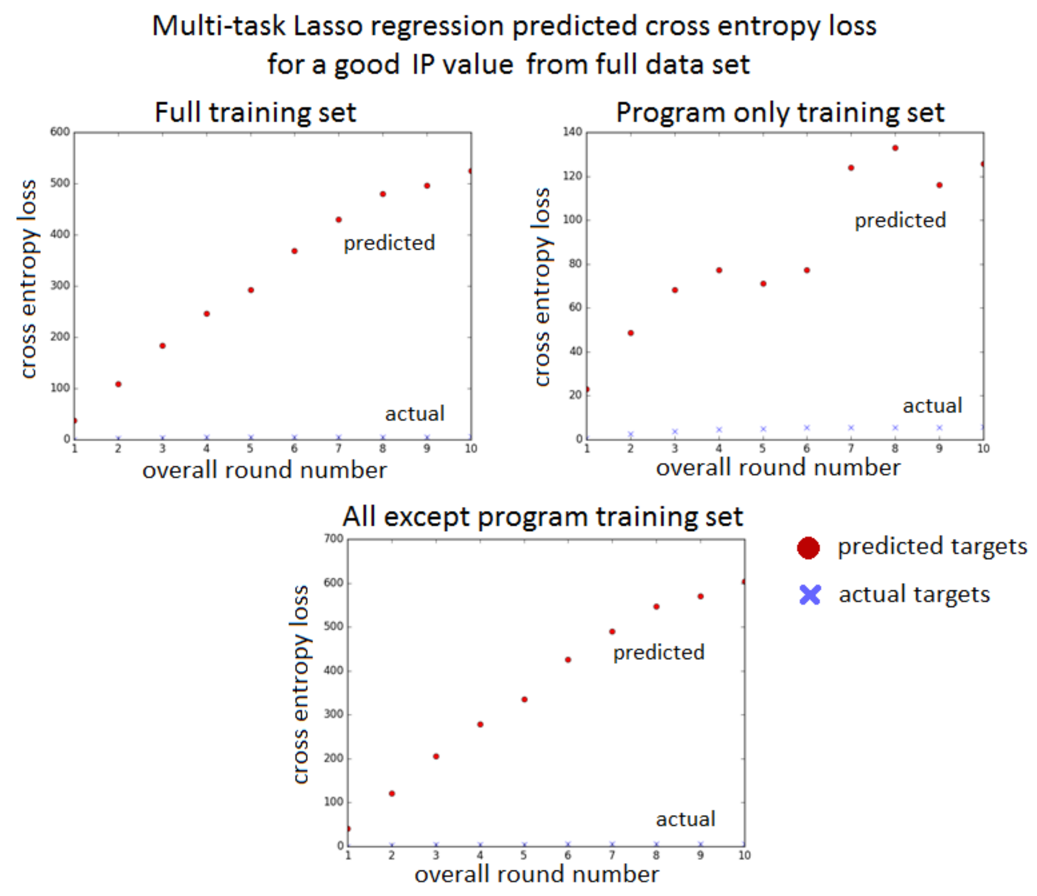

**Figure 41:** Predicted and actual cross entropy loss values for the good IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 351.29$, $r^2 = -48469.83$; Program only: $RMSE = 88.53$, $r^2 = -3077.57$; All except program: $RMSE = 402.56$, $r^2 = -63649.24$

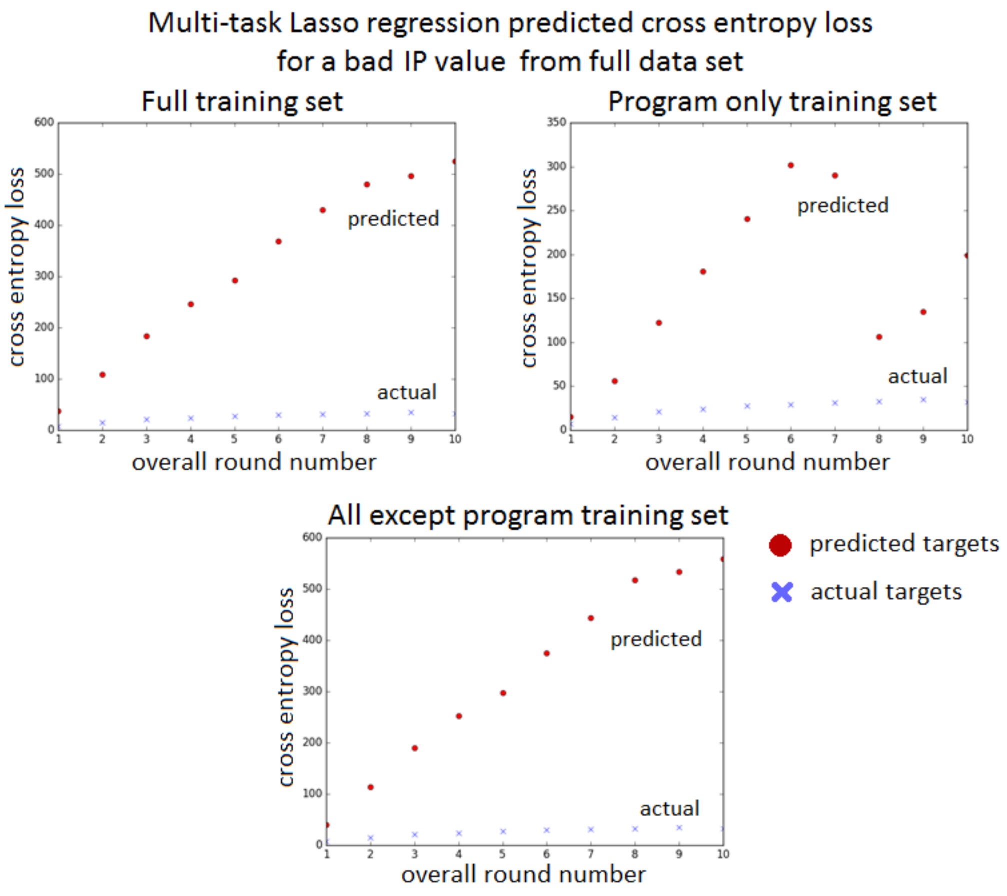

**Figure 42:** Predicted and actual cross entropy loss values for the bad IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 329.50$, $r^2 = -1504.04$; Program only: $RMSE = 163.23$, $r^2 = -368.36$; All except program: $RMSE = 348.53$, $r^2 = -1682.97$

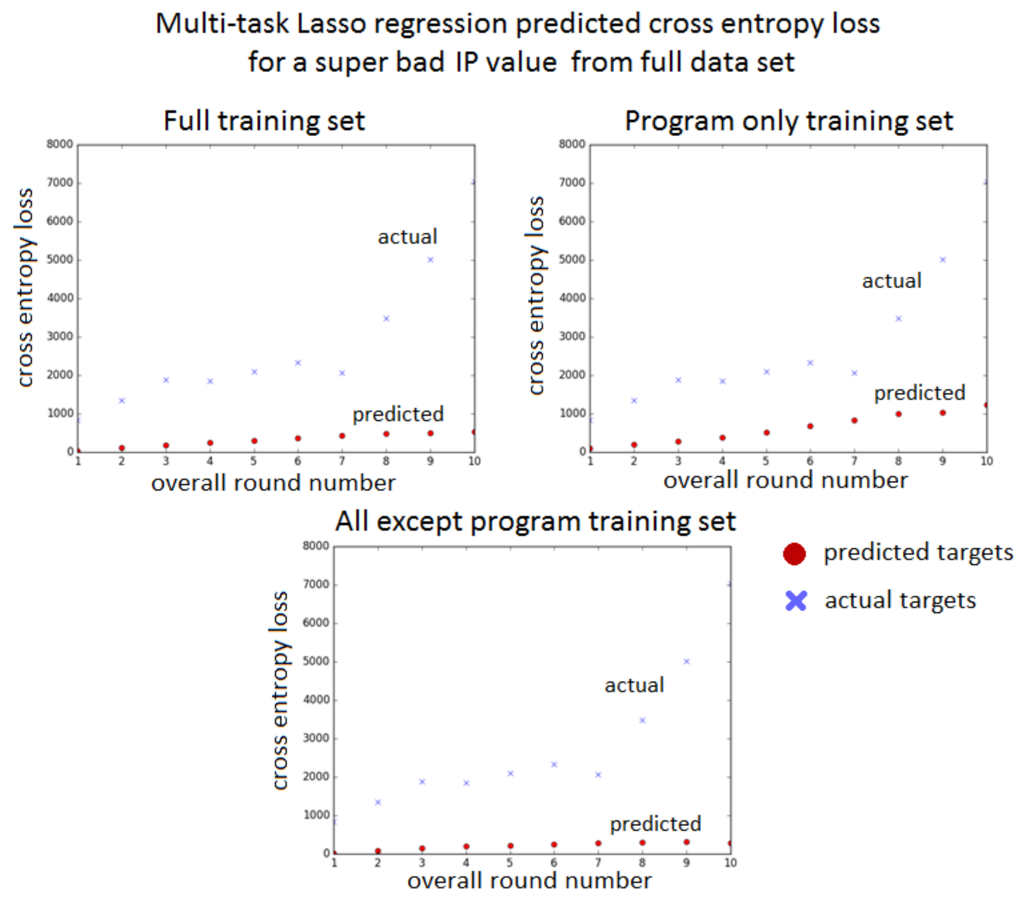

**Figure 43:** Predicted and actual cross entropy loss values for the super bad IP value and program trained on three different types of training sets. This comes from the full data set. The RMSE and $r^2$ values for each training set are: Full training set: $RMSE = 2987.38$, $r^2 = -1.77$; Program only: $RMSE = 2625.60$, $r^2 = -1.14$; All except program: $RMSE = 3115.73$, $r^2 = -2.01$

## 5.6 Limitations

One of the main limitations on the machine learning models in this project was the variation of the programs in the training set. We were able to gather data only from seven different programs. While these programs vary in their loop and call graph structures and memory usage, they are limited in that they can only use diet libc and are not allowed to dynamically allocate memory on the heap. This is due to current limitations in ASC's own implementation. This affects the performance of the machine learning models because it means that we do not see as much variability as we would like in the programs that we use to train. This may contribute to the result that the Bayesian ridge regression and multi-task Lasso regression performed considerably worse when tested on a new program that had never been seen in the training set.

Another limitation in this project was computational time and power. When running the Gaussian process model, we were unable to run it on the full data set and have it converge within a reasonable amount of time. Perhaps in future work, we could try to run a distributed implementation of the Gaussian process model on larger data sets.

# 6 Conclusion

We have presented a method of using machine learning to predict whether a given IP value and program will achieve speedup when run with ASC. We do this by predicting the cross entropy loss value that ASC outputs every time it encounters the given IP value in the given program.

With a good performance prediction framework, we can predict whether a given IP value and program will achieve speedup with ASC without ever having seen that program before. With the machine learning models presented in this project, we still have not achieved this holy grail. However, even though our machine learning models have not reached the performance level required to predict ASC speedup in practice, we still provide a useful framework for feature extraction and machine learning based performance prediction. Our results can also be used to develop better models in the future. After all, some of our models were quite promising - the Gaussian process could achieve the same performance when the test program was not included in the training set as when it was included.

## 6.1 Future work

One main avenue for future work is exploring more machine learning models that may better capture the relationship between program features and ASC performance. One model that can more accurately represent non-linear relationships between inputs and outputs is a neural network. In neural network regression, the inputs pass through multiple layers of logistic regressions before a final layer of linear regression. Between the regression layers, neural networks can also introduce hidden layers, which involve passing the input through a non-linear activation function [4]. This composition of regressions and hidden layers can be more successful than a single linear regression at modeling complicated non-linear relationships between inputs and outputs.

Also, we only tried one multi-task regression model in this project, but there are actually many different ways of implementing multi-task regression, including ones that can model non-linearity using neural networks. Thus, it would be an interesting point of future work to try both single-task and multi-task neural network regression models to predict cross entropy loss values for programs.

Another promising way to model the problem of predicting the performance of a program with ASC is to formulate it as a classification problem instead of a regression problem. In this project, we predicted actual cross entropy loss values for all programs and IP values. However, we could also predict whether a program and IP value would achieve "good", "bad", or "super bad" performance with ASC. While this is a less fine-grained analysis of the performance of each program, a coarser classification model may end up being more accurate than a more ambitious regression model.

In addition to the performance accuracy of a machine learning model, the interpretability of a machine learning model is also particularly important when predicting program performance with ASC. An interpretable model can help us identify the features of a program and IP value that actually make them achieve speedup with ASC. This can provide valuable heuristics for evaluating and improving the underlying models within ASC itself.

Machine learning models can vary greatly in interpretability. For example, neural networks and Gaussian process models can model complicated relationships but can be difficult to interpret [21]. Beyond these, there are some models that are inherently designed to have high interpretability. One such model is the Bayesian rule list. The Bayesian rule list is a model for classification that encodes the decision process as a series of if/then statements [22]. Once this model is successfully fitted to a data set, it naturally provides a decision list that is easy for humans to interpret. This advantage of interpretability would make Bayesian rule list classification an interesting avenue for further exploration with performance prediction on ASC.

Clearly, there are many ways to build on the machine learning framework presented in this project. Ultimately, with an accurate enough performance prediction framework, we not only solve the problem of IP selection within ASC, but we also establish a model that any user can use to evaluate whether their own programs will run well with ASC. In general, most users are not yet comfortable with ASC's approach to autoparallelization. ASC's use of machine learning to achieve autoparallelization is novel, and not much evaluation has been done on this method compared to more traditional autoparallelization methods. Thus, with this project, we have presented an important step forward in the evaluation of ASC as a method of autoparallelization, and we contribute to establishing ASC as a standard and useful method of autoparallelization in the context of other traditional autoparallelization methods.

# References

[1] A. Waterland, E. Angelino, R. P. Adams, J. Appavoo, and M. Seltzer, "Asc: Automatically scalable computation," *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[2] T. M. Heskes and B. Kappen, "On-line learning processes in artificial neural networks," in *Mathematical Approaches to Neural Networks* (J. Taylor, ed.), Amsterdam, Netherlands: Elsevier, 1993.

[3] S. Eldridge, J. Appavoo, , A. Joshi, A. Waterland, and M. Seltzer, "Towards general-purpose neural network computing," *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[4] K. P. Murphy, *Machine Learning: A Probabilistic Perspective.* Cambridge, MA, USA: MIT Press, 2012.

[5] M. Solnon, S. Arlot, and F. Bach, "Multi-task regression using minimal penalties," *Journal of Machine Learning Research*, vol. 13, pp. 2773–2812, 2012.

[6] R. Bagrodia, E. Deelman, S. Docy, and T. Phan, "Performance prediction of large parallel applications using parallel simulations," *Symposium on Principles and Practice of Parallel Programming*, 1999.

[7] J. Zhai, W. Chen, W. Zheng, and K. Li, "Performance prediction for large-scale parallel applications using representative replay," *IEEE Transactions on Computers*, 2014.

[8] V. Adve and M. K. Vernon, "A deterministic model for parallel program performance evaluation," *Technical Report CS-TR98-333, Computer Science Dept., Rice University*, 1998.

[9] G. Zheng, G. Gupta, E. Bohm, I. Dooley, and L. V. Kale, "Simulating large scale parallel applications using statistical models for sequential execution blocks," *International Conference on Parallel and Distributed Systems (ICPADS)*, 2010.

[10] K. Singh, E. Ipek, S. A. McKee1, B. R. de Supinski, M. Schulz, and R. Caruana, "Predicting parallel application performance via machine learning approaches," *Concurrency Computation: Practice and Experience*, 2007.

[11] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee, “Machine learning based online performance prediction for runtime parallelization and task scheduling,” *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[12] V. Adve and R. Sakellariou, “Compiler synthesis of task graphs for parallel program performance prediction,” *International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, 2000.

[13] E. Park, L.-N. Pouchet, , J. Cavazos, A. Cohen, and P. Sadayappan, “Predictive modeling in a polyhedral optimization space,” *International Symposium on Code Generation and Optimization (CGO)*, 2011.

[14] W. Killian, R. Miceli, E. Park, M. A. Vega, and J. Cavazos, “Performance improvement in kernels by guiding compiler auto-vectorization heuristics,” *Partnership for Advanced Computing in Europe (PRACE)*, 2014.

[15] K. Stock, L.-N. Pouchet, and P. Sadayappan, “Using machine learning to improve automatic vectorization,” *ACM Transactions on Architecture and Code Optimization*, 2012.

[16] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O’Boyle, and O. Temam, “Rapidly selecting good compiler optimizations using performance counters,” *Code Generation and Optimization (CGO)*, 2007.

[17] H. Leather, E. Bonilla, and M. O’Boyle, “Automatic feature generation for machine learning-based optimising compilation,” *ACM Transactions on Architecture and Code Optimization (TACO)*, 2014.

[18] E. Park, J. Cavazos, and M. A. Alvarez, “Using graph-based program characterization for predictive modeling,” *International Symposium on Code Generation and Optimization (CGO)*, 2012.

[19] G. Yan, N. Brown, and D. Kong, “Exploring discriminatory features for automated malware classification,” *Detection of Intrusions and Malware, and Vulnerability Assessment*, 2013.

[20] 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*, vol. 12, pp. 2825–2830, 2011.

[21] O. Intrator and N. Intrator, “Interpreting neural-network results: a simulation study,” *Computational Statistics  Data Analysis*, vol. 37, no. 3, pp. 373–393, 2001.

[22] B. Letham, C. Rudin, T. H. McCormick, and D. Madigan, “Interpretable classifiers using rules and bayesian analysis: Building a better stroke prediction model,” *The Annals of Applied Statistics*, vol. 9, pp. 1350–1371, 2015.