Cost Model Mining of Java Bytecode

by

Aniket Kumar Lata

B.E., University of Mumbai 2012

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Electrical and Computer Engineering

2016

This thesis entitled:
Cost Model Mining of Java Bytecode
written by Aniket Kumar Lata
has been approved for the Department of Electrical and Computer Engineering

Prof. Pavol Černý

Prof. Ashutosh Trivedi

Prof. Sriram Sankaranarayanan

Date

Lata, Aniket Kumar (Electrical and Computer Engineering)

Cost Model Mining of Java Bytecode

Thesis directed by Prof. Pavol Černý

We consider the problem of developing an efficient cost model for Java Bytecode, where cost is determined in terms of execution time. The key insight is that the execution time of Java applications can be predicted by modeling these applications based on the method invocation counts and their linear fitting to get an accurate estimate. Cost estimates for an application can be useful in determining parameters such as worst case execution time which can form a basis for static analyses and can be useful in distributed applications.

Dedication

To all my mentors, friends and family for their support and guidance.

Acknowledgements

Contents

Tables

Table

Figures

Chapter 1

Introduction

Accurate execution time measurement of Java bytecode instructions is an important estimate in determining the total execution time of Java applications. There are a number of factors that make the execution time of Java bytecode difficult to predict. Java applications are run on a Java Virtual Machine (JVM) which translates Java bytecode to platform dependent machine code. It consists of stages that introduce optimizations in the translated machine code. A JIT Compilation further introduces non determinism in the execution of a program. Dynamic linking and loading is another factor that slows down the compilation process of Java. C/C++ code is typically compiled to an object file then multiple object files are linked together to produce a usable artifact such as an executable or dll. During the linking phase symbolic references in each object file are replaced with an actual memory address relative to the final executable. In Java, this linking phase is performed dynamically at runtime. When a Java class is compiled, all references to variables and methods are stored in the class's constant pool as a symbolic reference. The JVM implementation can choose when to resolve symbolic references, this can happen when the class file is verified, after being loaded, called eager or static resolution, instead this can happen when the symbolic reference is used for the first time called lazy or late resolution.

Due to these factors making Java programs less deterministic, it is useful to have a prediction mechanism that can provide a timing estimate for a block of instructions. My Master's thesis builds upon this idea of timing basic blocks and predicting the execution time of applications that use this basic blocks. A cost model has been developed that can efficiently predict the execution time of basic blocks and methods declared within a program. Fetching costs for methods used in popular libraries can be a very useful measurement in profiling of Java programs.

The cost model provides estimates pertaining to the execution time for expensive methods in Java bytecode. Java applications are profiled to generate:

(1) Counts of blocks/method invocations and

(2) Total execution time of the application

This profiling is performed with the help of bytecode instrumentation. The data generated from these runs is fed to a Linear Regression toolbox to efficiently predict the "time per basic block" or the "time per method" metric for that specific application.

The cost model tool can help in determining the time taken for each method or basic block to execute within an application. The worst case execution time (WCET) of a program is an important parameter which helps in real time scheduling applications. WCET can be predicted with the help of cost information from the model. The costs can be fed to a static analyses framework which can check all possible path combinations to get a WCET metric for that application. Cost modeling can also be used to detect security vulnerabilities in Java applications. Security vulnerabilities can be classified broadly into Availability and Confidentiality attacks. Availability problems arise when a user-provided input controls the asymptotic complexity or termination of a program component. The cost model can flag the blocks within a program that consume more CPU time with a high cost. A section of code susceptible to availability vulnerabilities can be determined by checking cost annotations of each block and what kind of inputs would reach this block of code.

An important use of the cost model is to predict the execution time of Java applications whose libraries are profiled with the cost model. The model data forms a training input set and the application data forms a testing input set. The execution time of each method or basic block is an output from the training data set. These estimates are applied to the testing data set to predict the execution time of this application for a specific set of inputs.

Chapter 2

Motivating Examples and the Cost Model approach

2.1 Basic Block Timing

Java bytecode can be profiled at different points of interest to measure execution time for these sections of code. These measurements need to be consistent to ensure disambiguation of the timing analysis. To perform these measurements, we use instrumentation techniques for adding bytecode to existing applications. Bytecode instrumentation is a process where new functionality is added to a program by modifying the bytecode of a set of classes before they are loaded by the virtual machine. The JVM interprets that bytecode at runtime and then executes the correct native system instructions according to that bytecode.

We start our analysis by trying to disambiguate the execution time for basic blocks within a Java application. These are the most fundamental units of a program. Basic blocks present a straight-line code sequence making them highly amenable to analysis. These blocks have a singe entry point and a single exit point. Basic Blocks are useful in numerous program transformations and optimizations.

Our aim in this section is to determine the time taken by basic blocks of a Java application to execute for a given set of inputs. One very straightforward approach in timing basic blocks is to identify the blocks and add timing code before and after the blocks. This analysis is termed as "Fine grain analysis". In the next subsection, we present experiments conducted with fine grain analysis, the results obtained and subsequent inferences.

2.1.1 Fine grain analysis

Java bytecode can be instrumented to add source code for timing the basic blocks within a program. We add System.nanotime() calls before and after each basic block and print the time measurement for each basic block. Our goal here is to verify if this method works accurately for execution time disambiguation of basic blocks. For disambiguation profiling, we consider an example program shown below with three basic blocks.

2.1.1.1 Examples

```
public void workerFunction(int arg){
        int N = arg;
    int x = 0, a = 1, m = 0, i = 0;
    ArrayList<Integer> alist = new ArrayList<Integer>();

    if(N > 50) {
                // Basic block 1 starts
        x++;
        x = x * N;
        x &= (a << 1);
        alist.add(x);
        // Basic block 1 ends
     }  else {
                        do{
                // Basic block 2 starts
                        x += i;
                m = (x >> (N - i) & 1);
                i++;
                alist.add(x);
                // Basic block 2 ends
             }while(i < N);
        }
        if(x % 2 == 0) {
                        // Basic block 3 starts
            x = x * N;
            x %= N/3.14;
            x |= 34;
            java.lang.Thread.sleep(1);
            // Basic block 3 ends
        }
 }
```

The worker function is instrumented to measure execution time of the three basic blocks as shown in the modified worker function:

```
public void workerFunction(int arg){
       int N = arg;
    int x = 0, a = 1, m = 0, i = 0;
    ArrayList<Integer> alist = new ArrayList<Integer>();

    if(N > 50) {
              // Basic block 1 starts
              long l1 = System.nanoTime();
       x++;
       x = x * N;
       x &= (a << 1);
       alist.add(x);
       long l4 = System.nanoTime();
       long l7 = l4 - l1;
       double d = (double)l7 / 1000000D;
       System.out.println((new StringBuilder()).append("\n\rDURATION: ").append(d
          ).toString());
       // Basic block 1 ends
     }  else {
                     do{
              // Basic block 2 starts
                            long l2 = System.nanoTime();
                     x += i;
              m = (x >> (N - i) & 1);
              i++;
              alist.add(x);
              long l6 = System.nanoTime();
              long l9 = l6 - l2;
              double d2 = (double)l9 / 1000000D;
              System.out.println((new StringBuilder()).append("\n\rDURATION: ").
                 append(d2).toString());
              // Basic block 2 ends
           }while(i < N);
       }
       if(x % 2 == 0) {
                     // Basic block 3 starts
                     long l3 = System.nanoTime();
          x = x * N;
```

```
            x %= N/3.14;
            x |= 34;
            java.lang.Thread.sleep(1);
            long l5 = System.nanoTime();
            long l8 = l5 - l3;
            double d1 = (double)l8 / 1000000D;
            System.out.println((new StringBuilder()).append("\n\rDURATION: ").
               append(d1).toString());
            // Basic block 3 ends
        }
}
```

2.1.1.2 Results

The basic block measurements were tested by executing the instrumented version of the code multiple times with the same input.

Commands: $ java -jar Test.jar 8

$ java -jar Test.jar 56

Run 1:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.440963

Basic block 2:

    Iteration 1: 0.332355

    Iteration 2: 0.0012

    Iteration 3: 0.001012

    Iteration 4: 0.000965

    Iteration 5: 0.000966

    Iteration 6: 0.000887

    Iteration 7: 0.000926

    Iteration 8: 0.000914

Basic block 3: 1.11022

Run 2:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.318824

Basic block 2:

Iteration 1: 0.310898

Iteration 2: 0.001169

Iteration 3: 0.001046

Iteration 4: 0.000961

Iteration 5: 0.000945

Iteration 6: 0.000895

Iteration 7: 0.000904

Iteration 8: 0.000867

Basic block 3: 1.086663

Run 3:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.299579

Basic block 2:

Iteration 1: 0.373075

Iteration 2: 0.001417

Iteration 3: 0.001294

Iteration 4: 0.00115

Iteration 5: 0.001131

Iteration 6: 0.001112

Iteration 7: 0.001073

Iteration 8: 0.001084

Basic block 3: 1.089375

Run 4:

Execution time measurements in milliseconds(ms):

Basic block 1: 0.310518

Basic block 2:

Iteration 1: 0.317869

Iteration 2: 0.001171

Iteration 3: 0.001075

Iteration 4: 0.000916

Iteration 5: 0.001008

Iteration 6: 0.000965

Iteration 7: 0.000921

Iteration 8: 0.000934

Basic block 3: 1.095043

Statistics:

Basic block 1:	Mean: 0.34247	Standard Deviation: 0.06613	Variance: 0.00437
Basic block 2:	Mean: 0.0426	Standard Deviation: 0.11206	Variance: 0.01256
Basic block 3:	Mean: 1.09533	Standard Deviation: 0.01053	Variance: 0.00011

2.1.1.3 Inferences

Disambiguation among the execution time measurements can be ensured if the timing measurements are consistent. From the above measurements, it can be seen that the variance is significant considering the fact that timing measurements require very high accuracy. Basic block 2 has a large standard deviation from the mean taking into account all the four runs of this application. It should be noted that the input argument was the same for each run. Basic block 1 is reachable with inputs greater than 50 and basic block 2 is reachable when input argument is less than 50. The statistics show that even though the inputs did not vary, there is significant amount of variance in the timings.

2.1.2 Disambiguation

The fine grain approach does not work in disambiguation of execution time for basic blocks as the standard deviation of the reading is significant when multiple runs are accounted. We propose a novel approach to disambiguate execution time for Java applications. The cost model profiles a program to measure the execution time for one entire run as opposed to timing individual sections of code. Instrumentation is added to fetch block invocation counts.

The basic example would be profiled as shown in the subsequent sections to generate a cost model.

2.1.2.1 Execution time Profiling

```
public void workerFunction(int arg){
        // Instrumentation for exec time
        long startTime = System.nanoTime();
        int N = arg;
    int x = 0, a = 1, m = 0, i = 0;
    ArrayList<Integer> alist = new ArrayList<Integer>();

    if(N > 50) {
                // Basic block 1 starts
        x++;
        x = x * N;
        x &= (a << 1);
        alist.add(x);
        // Basic block 1 ends
     }  else {
                        do{
                // Basic block 2 starts
                        x += i;
                m = (x >> (N - i) & 1);
                i++;
                alist.add(x);
                // Basic block 2 ends
             }while(i < N);
        }
        if(x % 2 == 0) {
                        // Basic block 3 starts
            x = x * N;
            x %= N/3.14;
            x |= 34;
            java.lang.Thread.sleep(1);
            // Basic block 3 ends
        }
      long endTime = System.nanoTime();
      long elapsedTime = endTime - startTime;
      double duration = (double)elapsedTime / 1000000.0;
      System.out.println(duration);
 }
```

2.1.2.2 Block invocation Profiling

```
public void workerFunction(int arg){
       // Instrumentation for block invocation counts
       // counters: counter_bb1, counter_bb2, counter_bb3
       int counter_bb1 = 0, counter_bb2 = 0, counter_bb3 = 0;
       int N = arg;
    int x = 0, a = 1, m = 0, i = 0;
    ArrayList<Integer> alist = new ArrayList<Integer>();

    if(N > 50) {
              // Basic block 1 starts
       x++;
       x = x * N;
       x &= (a << 1);
       alist.add(x);
       counter_bb1++;
       // Basic block 1 ends
     }  else {
                     do{
              // Basic block 2 starts
                     x += i;
              m = (x >> (N - i) & 1);
              i++;
              alist.add(x);
              counter_bb2++;
              // Basic block 2 ends
            }while(i < N);
       }
       if(x % 2 == 0) {
                     // Basic block 3 starts
          x = x * N;
          x %= N/3.14;
          x |= 34;
          counter_bb3++;
          java.lang.Thread.sleep(1);
          // Basic block 3 ends
       }
}
```

Model generation requires two passes for each input: execution time pass and block invocation count pass. This data is logged on the disk for a large number of inputs.

2.1.2.3 Results

The cost model disambiguation approach was tested on the same basic blocks example. The inputs to the application were randomly generated. Below are the results obtained:

Output 1: Number of runs consists of 800 distinct inputs

Coefficients:

workerFunction_basicBlock1: 0.2212857

workerFunction_basicBlock2: 0.0095599

workerFunction_basicBlock3: 1.2227574

Output 2: Number of runs consists of 1600 distinct inputs

Coefficients:

workerFunction_basicBlock1: 0.2438942

workerFunction_basicBlock2: 0.0096788

workerFunction_basicBlock3: 1.2122846

Output 3: Number of runs consists of 2400 distinct inputs

Coefficients:

workerFunction_basicBlock1: 0.2099943

workerFunction_basicBlock2: 0.0088905

workerFunction_basicBlock3: 1.2196536

Output 4: Number of runs consists of 3200 distinct inputs

Coefficients:

workerFunction_basicBlock1: 0.2156011

workerFunction_basicBlock2: 0.0089118

workerFunction_basicBlock3: 1.2254903

Statistics:

Basic block 1:	Mean: 0.22269	Standard Deviation: 0.01487	Variance: 0.00022
Basic block 2:	Mean: 0.00926	Standard Deviation: 0.00042	Variance: 0
Basic block 3:	Mean: 1.22005	Standard Deviation: 0.0057	Variance: 0.00003

2.1.2.4 Inference

The cost model disambiguation approach shows that there is minimal standard deviation for each basic block timing estimate, much lower than the fine grain analysis approach. As it can be seen, the variance is 0 for basic block 2 whereas the fine grain approach showed much higher variance for this basic block.

2.2 Bytecode Instruction disambiguation

The cost model approach can be applied at the bytecode level to disambiguate different types of bytecode instructions and their execution times on a target platform. In this example, the disambiguation of bytecode instructions has been performed for an arithmetic intensive program.

2.2.1 Example

```
public class TestBinaryOp {
        public static void main(String[] args) {
                int x = Integer.parseInt(args[0]);
                int y = Integer.parseInt(args[1]);
                int numAdd = Integer.parseInt(args[2]);
                int numSub = Integer.parseInt(args[3]);
                int numMul = Integer.parseInt(args[4]);
                int numDiv = Integer.parseInt(args[5]);
                int result;

                for( int i = 0; i < numAdd; i++ ) {
                        result = x + y;
                }
                for( int i = 0; i < numSub; i++ ) {
                        result = x - y;
                }
                for( int i = 0; i < numMul; i++ ) {
                        result = x * y;
                }
                for( int i = 0; i < numDiv; i++ ) {
                        result = x / y;
                }
        }
}
```

2.2.2 Results

The bytecode instructions have been divided into different classes depending upon the correlation with one another.

ARITHMETIC - ADD,SUB,MUL,DIV,LOAD,STORE,INC,AND,OR,XOR,SHL,SHR,NEG

MEMORYALLOCATIONS - NEW,DUP,GETFIELD,PUTFIELD,PUSH,POP,GETSTATIC,LDC

IOCALLS - java.io.*

UTILCALLS - java.util.*

MATHCALLS - java.lang.Math.*

Below are the estimates for the expensive bytecode categories of instructions:

	Estimate	Std. Error
ArithmeticOp	1.098e-06	1.841e-08
MemoryAlloc	3.842e-03	7.779e-04

It can be seen here that the cost for arithmetic instructions and memory allocations has been estimated by the model. However, at the lowest level this disambiguation can tend to have some deviation depending on the interference from other instructions to the measurement of arithmetic and related bytecode costs. We move one level higher to groups of bytecode instructions - precisely basic blocks and methods to achieve better disambiguation.

2.3 Method disambiguation

We have seen that timing basic blocks for the same input produces inconsistent results. Our approach relies on timing the entire application as opposed to timing each fundamental unit of the program. With the cost model, we measure the execution time of the application and log method invocation counts for a number of inputs. Linear regression is used to produce the estimates for each method. The example given below illustrates our approach.

```
public class Functions {

        public void function1() throws InterruptedException{
                java.lang.Thread.sleep(1);
        }

        public void function2() throws InterruptedException{
                java.lang.Thread.sleep(10);
        }

        public void function3() throws InterruptedException{
                java.lang.Thread.sleep(500);
        }

        public void function4() throws InterruptedException{
                java.lang.Thread.sleep(100);
        }

        public void function5(){
                return;
        }

}
```

The ”Functions” library consists of five functions and each one takes different amounts of time to execute. Our goal here is to disambiguate these functions by accurately identifying the amount of time for which they run.

The cost model approach instruments the Functions library to fetch method invocation counts and total execution time for the application. The cost model is explained in detail in the later sections. A test application has been written that calls the methods in the library with different call distributions. As it can be seen in the test class below, methods are invoked in different order for different inputs received from the users. This exercise simulates a good coverage of the functions being called in random distributions.

2.3.1 Basic example

```
public class MainFunc {

        public static void main(String args[]) throws InterruptedException{

                int N = Integer.parseInt(args[0]);
                Functions func = new Functions();

                switch(N){
                        case 1:
                                func.function1();
                                func.function3();
                                func.function3();
                                func.function5();
                                func.function3();
                                break;
                        case 2:
                                func.function5();
                                func.function2();
                                func.function2();
                                func.function4();
                                func.function4();
                                break;
                        case 3:
                                func.function3();
                                func.function4();
                                func.function4();
                                func.function5();
                                func.function4();
                                break;
                        case 4:
                                func.function1();
                                func.function1();
                                func.function5();
                                func.function4();
                                func.function2();
```

```
                                        break;
                                case 5:
                                        func.function1();
                                        func.function1();
                                        func.function1();
                                        func.function2();
                                        func.function1();
                                        break;
                                case 6:
                                        func.function5();
                                        func.function4();
                                        func.function3();
                                        func.function2();
                                        func.function1();
                                        break;
                                }
                }
}
```

2.3.2 Initial Results

Multiple linear regression is used to generate the best fitting estimates that can disambiguate the time taken by each method. The dependent variable is ”execution time” for a specific run and the independent variables are ”method names” for that same run. The estimates are generated for each independent variable which are the ”functions” in the above example. The estimates generated using our cost model tool for 200 inputs are as shown below:

	Estimate
Function1	1.131e+00
Function2	1.014e+01
Function3	5.002e+02
Function4	1.002e+02
Function5	2.807e-02

As it can be seen, the estimates generated are quite accurate. We can compare the estimates to the sleep call parameters within each function as the functions are not doing any other work. These estimates become more and more precise as the input coverage gets better and the number of test inputs increases.

Thus, it can be inferred from this experiment that the method disambiguation experiment worked well using the cost model for the basic example illustrated.

2.3.3 Problems with method disambiguation

Method disambiguation worked well in the example demonstrated in section 2.2.1. The methods used in the library did not have any control flows. If the method is a sequential block of statements, for a single input the same section of code will be executed. However with control flows being a part of the method, different branches are executed for differing inputs to the program. This can be illustrated well by modifying the basic example presented earlier.

Control flows introduce path imbalances in the program execution. In such cases, it becomes difficult to provide an estimate for the execution time of such methods if the execution time of paths taken for an input is noticeably different.

2.3.3.1 Control flow example

```
public void function1() throws InterruptedException{
        java.lang.Thread.sleep(20);
}

public void function2(int n) throws InterruptedException{
        java.lang.Thread.sleep(5);
        if(n > 6)
                function4();
}

public void function3() throws InterruptedException{
        java.lang.Thread.sleep(500);
}

public void function4() throws InterruptedException{
        java.lang.Thread.sleep(100);
}

public void function5(){
        return;
}
```

2.3.3.2 Results

The control flow example in section 2.3.1 is a modification of the basic example shown with methods. We have a branch in function2() which changes the predicted estimates significantly. It can be seen that function5() gets a negative estimate and the estimate for function4() is incorrect. function2() has a branch and it's estimate will vary depending upon the value of 'n'. It is incorrect to have an estimate for functions where the control flow determines the execution time of the program.

	Estimate
function1	21.323
function2	39.599
function3	500.569
function4	5.361
function5	-0.907

The aim of this example was to illustrate the amount of inaccuracy that a branch can introduce within the cost measurements. Real life programs will have a large number of branches and if these branches introduce paths that are significantly different from one another in terms of execution time, it is necessary to mitigate the effects of control flows within a program.

2.4 Cost Model

We have tested the cost model approach on instruction, basic block and method level. It could be seen that on the instruction and method level, the standard error with our estimates is high due to the presence of correlation between our independent variables in regression. In the following example, we highlight the problem of correlation between buckets.

2.4.1 Correlation

Let us have a look at the following Modular exponentiation class. It consists of three basic blocks.

```
public class ModularExponentiation {
       public static void main(String args[]) throws InterruptedException {
              Integer x = Integer.parseInt(args[0]);
       Integer e = Integer.parseInt(args[1]);
       Integer N = Integer.parseInt(args[2]);
              squareMult(x, e, N);
       }

       public static Integer squareMult(Integer x, Integer e, Integer N) throws
          InterruptedException{
              ArrayList<Integer> alist = new ArrayList<Integer>();
       Integer y = 1;
       int n = Integer.SIZE;
       int i = n;
       do {
                            // basic block 1 starts
                     y = y * y;
              y = y % N;
              //basic block 1 ends
              if(((e >> (n-i)) & 1) == 1){
                            // basic block 2 starts
                            y = y * x;
                  java.lang.Thread.sleep(10);
                  y = y % N;
                  // basic block 2 ends
              }
              // basic block 3 starts
              i--;
              alist.add(y);
              // basic block 3 ends
          } while(i >= 1);
              return y;
    }
}
```

On running our cost model over this program for 800 distinct inputs, we get the below estimates for basic blocks 1,2 and 3.

Coefficients:

Basic block 1: 0.01217091

Basic block 2: 10.10931665

Basic block 3: NA

It can be clearly seen that basic blocks 1 (bb1) and 3 (bb3) are executed each time and hence the block invocation count for these blocks will be 1 for all the executions. The execution of basic block 2 is dependent on a condition. It can be said that there is a high correlation between basic blocks 1 and 3 such that every time bb1 is executed, bb3 is also executed and vice versa. Our cost model is unable to disambiguate between basic blocks 1 and 3 from the statistics that we obtain after profiling. The profiling procedure is explained in detail in the Implementation section. The estimate obtained for bb1 is actually the estimate for bb1 and bb3 combined. The linear regression algorithm accounts for only one of these two independent variables and generates an estimate for this variable.

In the next section, we will present some techniques that can help refine the cost model by reducing the effect of correlation and thus, the standard error for each estimate.

Chapter 3

Model

Our cost model needs to predict the value of the dependent variable – execution time, using a linear function of the independent variables – method counts.

A naive approach to bucketing would be accounting for each method used in the application as a separate bucket. The standard error for coefficients in multiple regression can be high if we don't account for the correct subsets of independent variables in the model. The variance of coefficient for a variable could increase if there is high correlation among with another variable. The mean squared error for all the variables gives a good indication of the estimate provided by the cost model. Even though a model shows an ideal coefficient of determination, it may not be sufficient to give an accurate estimate for an independent test application using the same independent variables i.e. methods in our case. There is a need to refine the cost model to address the standard error among variables within a model and the mean squared error for accurate estimation of test applications.

Model refinement addresses two specific problems: **Subset selection** and **Estimation Accuracy**.

3.1 Subset Selection

An important decision to be made is the bucketing (categorization) of methods in a Java application to perform multiple linear regression. This is termed as "subset selection" for the model. The naive approach considers all independent variables in the data for prediction. There are several reasons why this could be undesirable:

- Estimates of regression coefficients are likely to be unstable due to multi-collinearity in models with many variables.
- We get better insights into the influence of regressors from models with fewer variables as the coefficients are more stable for parsimonious models.
- It can be shown that using independent variables that are uncorrelated with the dependent variable will increase the variance of predictions.
- It can be shown that dropping independent variables that have small (non-zero) coefficients can reduce the average error of predictions.

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \cdots + \beta_n X_n \tag{3.1}$$

$$MSE = \frac{1}{N} \sum_{i=0}^{N} (y_i - \hat{y}_i)^2 \tag{3.2}$$

Chapter 4

Algorithm

Chapter 5

Implementation

Chapter 6

Experiments

Chapter 7

Conclusion