ABSTRACT

REALIZATION OF MODEL-DRIVEN ENGINEERING FOR BIG DATA:
A BASEBALL ANALYTICS USE CASE

by Kaan Tamer Koseler

Data collection and analysis is widespread across all industries, leading to a glut of data and a dearth of specialists who can use this data to derive insights. Accompanying the new “Big Data” paradigm is a resurgence in interest in machine learning techniques. Using machine learning techniques to work with "Big Data" is a complex task, often requiring specialized knowledge of the problem space as well as appropriate computer algorithms and approaches. However, such specialists who also possess programming ability are difficult to find and expensive to train. The gap between the problem space and the software solution often includes developers who lack the requisite domain-specific knowledge. The Model-Driven Engineering (MDE) paradigm helps close this gap by allowing developers to implement quality software by modeling it using high-level domain specific concepts. In this thesis, we attempt to demonstrate the plausibility of applying MDE to big data by considering a use case of machine learning baseball analytics, specifically, prediction of the next pitch. We model and implement MDE solutions to this use case by employing and updating an existing, but untested, Domain-Specific Modeling Language (DSML). We implement model instances considering different prediction factors and a code generation scheme for this DSML that is targeted at a binary classification problem of fastball versus non-fastball. Our goal is to help demonstrate the viability of the MDE paradigm in the machine learning domain, make machine learning software development more accessible and formalized, and help facilitate future research in this area.
REALIZATION OF MODEL-DRIVEN ENGINEERING FOR BIG DATA:
A BASEBALL ANALYTICS USE CASE

Thesis

Submitted to the
Faculty of Miami University
in partial fulfillment of
the requirements for the degree of
Master of Science in Computer Science
by
Kaan Tamer Koseler
Miami University
Oxford, Ohio
2018

Adviser: Dr. Matthew Stephan
Reader: Dr. James Kiper
Reader: Dr. Michael Zmuda

©2018 Kaan Tamer Koseler
This thesis titled

REALIZATION OF MODEL-DRIVEN ENGINEERING FOR BIG DATA: A BASEBALL ANALYTICS USE CASE

by

Kaan Tamer Koseler

has been approved for publication by

College of Engineering and Computing

and

Department of Computer Science and Software Engineering

__________________________________________

Dr. Matthew Stephan

__________________________________________

Dr. James Kiper

__________________________________________

Dr. Michael Zmuda
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2 Threats to Validity</td>
<td>32</td>
</tr>
<tr>
<td>5.3 Lessons Learned and Challenges</td>
<td>33</td>
</tr>
<tr>
<td>5.4 Potential Impact and Future Work</td>
<td>34</td>
</tr>
<tr>
<td>6 Related Work</td>
<td>36</td>
</tr>
<tr>
<td>7 Conclusion</td>
<td>38</td>
</tr>
<tr>
<td>Glossary</td>
<td>40</td>
</tr>
<tr>
<td>Bibliography</td>
<td>42</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Simple example of a classification problem ........................................... 4
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>The basic 3+1 model architecture [20]</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>The probability distribution P(A,B,C) [14]</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Breuker’s DSML syntax [14]</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Our Approach</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>The metamodel in Papyrus</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>Basic Model Instance with Count as Sole Factor (Strikes and Balls)</td>
<td>17</td>
</tr>
<tr>
<td>3.4</td>
<td>Less Basic Model Instance with Count and Pitchers ERA as factors</td>
<td>18</td>
</tr>
<tr>
<td>3.5</td>
<td>Invalid Model Instance - Factor Missing its Random Variable</td>
<td>19</td>
</tr>
<tr>
<td>3.6</td>
<td>Complete Model Instance with 18 Factors</td>
<td>20</td>
</tr>
<tr>
<td>4.1</td>
<td>Prediction Accuracy for our Work and Ganeshapillai and Guttag’s [28]</td>
<td>30</td>
</tr>
</tbody>
</table>
List of Listings

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The code generation engine</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of automatically generated code</td>
</tr>
<tr>
<td>3.3</td>
<td>Python function to output properly formatted string</td>
</tr>
</tbody>
</table>
Acknowledgements

Let me begin by acknowledging the Computer Science department here at Miami. In particular, my advisor Dr. Matthew Stephan. As I look back, we accomplished quite a bit throughout our time working together. Thank you for always keeping me on track and productive. Of course, you did it all with a smile, eh? Your future students are in good hands, and I wish you and them nothing but the best. Let me also thank Dr. Jerry Gannod, who was my original advisor before leaving to become the Chair of the CS department at Tennessee Tech. You were the first to believe in me as I made the transition from psychology to computer science, and for that I will be forever grateful. Thank you for your confidence in my abilities and your sage advice. Also Dr. Bo Brinkman, currently at Google, who first offered up to me the possibility of joining this program. Your kindness and openness really meant a lot.

The entire department deserves my thanks. In particular I want to thank Dr. James Kiper and Dr. Michael Zmuda for serving on my committee, and for their valuable feedback as I’ve worked on this thesis. And Dr. Eric Bachmann, whom I have pestered endlessly, delaying countless cups of coffee. Thank you for always answering my questions and being willing to talk. Dr. DJ Rao, Dr. Hakam Alomari, Dr. Angel Bravo, Dr. John Femiani, Drew Foster, Norm Krumpe, Patty Strecker. I consider myself a lucky man to have met you all. Thank you for everything. I also want to thank my lab-mates for the endless laughter, companionship, and discussions about algorithms: Yeshwanth, Julius, Sunny, Ania, Gianna, Hawkar, and Alex. My time here would have felt considerably more empty without you all.

Let me also take some space to thank the Miami University Library system for allowing me to pursue a truly comprehensive education. There is a lot more to the world than just computer science, and the libraries allowed me to indulge every interest and whim. They were a real oasis when I needed to flee the data structures and algorithms.

To my girlfriend Liz, your constant love and support throughout this often frustrating thing we call graduate school is something I will never forget. I can’t thank you enough for all you’ve done for me. I certainly couldn’t have done it without you. To my dear friends back in New York: Vitch, Maurice, Greg, Andrew, Spencer, Eduardo, and Ashantiana, your long and steady friendship has
been priceless to me. People often say that you forget the friends you made in high school as you grow older. I’m lucky to say that I have absolutely no idea what they’re talking about.

Last but not least, I want to thank my family for helping me become the person that I am today. To my mother Dilek, you have always encouraged and loved me without asking anything in return. Thank you for all your hard work and support. None of this would be possible without you. To my father Taner, your unwavering support of my goals and your unshakable confidence in me has meant a great deal. Of course, I could write pages thanking the rest of my family, but you all know who you are. Thanks for everything. I was very fortunate to be surrounded by such loving people.
Statement of Originality

These statements certify that, to the best of my knowledge, the content of this thesis is my own work. This thesis has not been submitted for any other degree. I certify that the intellectual content of this thesis is the product of my own original work and that all the assistance received in preparing this thesis and sources have been acknowledged in the thesis.

Parts of Chapter 1 and 2 were published in a Journal paper in Applied Artificial Intelligence [1] and in a Miami University technical report [2].

Parts of Chapter 1, 2, and 3 were published in our paper in the 2017 International Summer School on Domain-Specific Modeling Theory and Practice [3].

We have submitted much of the original contents of this thesis for consideration to the 2018 International Conference on Model Driven Engineering Languages and Systems [4].
Chapter 1

Introduction

Using data to derive patterns is a critical driver of human knowledge and progress [5]. The recent emergence of the "big data" paradigm has had a substantial effect on this process. Due to the global proliferation of information processing devices and their ubiquity in the developed world, we have more data than ever before. This totality of data is both so immense and useful that it appears to be ushering in an entirely new era of human civilization: "The Information Age" [6]. "Big data" is a popular term capturing the essence of this era: There is far more data than humans can process without the assistance of computers and algorithms. Individuals or organizations that can leverage this data to derive insights are richly rewarded [7]. This almost always requires advanced data processing and analysis techniques. One particularly popular technique is machine learning. There is some controversy about what machine learning entails, but a generally accepted definition is finding patterns in data that "provide insight or enable fast and accurate decision making" [8]. This usually takes the form of an output of predictions on new examples.

The explosive growth of big data analytics and machine learning has introduced several new challenges for software engineers. Due to the somewhat esoteric nature of the field, it is difficult to find software engineers that can develop and maintain applications that incorporate machine learning techniques to analyze data [9]. Big data analytics is also undergoing rapid evolution and maturation, requiring engineers to keep abreast of new developments and avenues of research.

One possible approach to address these challenges is to utilize Model-Driven Engineering
MDE is a paradigm focused on formal abstractions to build a "model" of a particular application or software artifact [10]. These models abstract the underlying source code to facilitate better design and maintenance of software. Importantly, in the MDE paradigm, this model is used throughout the software engineering life cycle, from requirements to testing and deployment. Perhaps the most useful aspect of this paradigm is automatic code generation through interpretation of the model. Ideally, an engineer may progress through the software life cycle without ever manipulating source code, which is very low level and can be cumbersome to manage and maintain.

There exist different formal syntaxes through which these models are expressed. One common and widely-used example is the Unified Modeling Language [11]. UML is meant to be language-agnostic, and as a result it does not define a standard for automatic code generation. Engineers seeking to utilize such features usually must develop them in-house. Often engineers are developing software for highly specific domains and prefer to handle the code generation themselves. A domain may also require the development of its own formal modeling syntax. These are referred to as domain-specific modeling languages (DSML) [12].

For this project, we employ a machine learning DSML to a specific use case of Baseball analytics. Baseball is particularly suited for this type of analysis due to its rigidly discrete nature as well as the wealth of statistical information that has been compiled over its long history. Big data techniques are also increasingly being incorporated into baseball analytics [13]. Previous work by Breuker defines a syntax for a DSML that abstracts machine learning techniques [14]. Our project serves as the first-time realization of their syntax and language in a popular and appropriate domain, including the implementation of a code generation scheme for a Baseball binary classification problem.

We set out to answer the following questions,

1. Is it possible to encapsulate machine learning binary classification concepts in a DSML and accompanying code generation scheme?
   (a) What challenges exist in doing so?

2. Can we realize a complete application of Machine Learning MDE for big data baseball that allows for model updates and code generation?
Chapter 2

Background

This section describes background material for our research. In Section 2.1 we introduce general concepts of supervised machine learning and the binary classification problem. In Section 2.2 we provide a primer on Model-Driven Engineering and Domain-Specific Model Languages, as well as the foundation laid by Breuker in crafting a machine-learning DSML. In Section 2.3 we overview the field of Baseball Analytics and its intersection with machine learning.

2.1 Machine Learning

There is broad agreement that Machine Learning involves automated pattern extraction from data [15]. It is often the case that the patterns extracted from machine learning techniques are used to make predictions. Thus, the type of machine learning that is usually employed by analysts is referred to as supervised machine learning. There are other types such as unsupervised learning and reinforcement learning. But supervised learning is the dominant approach [15]. Our research focuses on this type of learning, which Bishop defines as consisting of problems that take in training data example input vectors, \( x_i \), and their corresponding target vectors, \( y_i \) [16]. For example, consider the case of predicting whether a certain student will gain admittance into Miami University. A natural place to begin is an examination of past admission cycles. We might take in input vectors of student attributes like GPA, SAT score, and admission status from the year 2017. The crucial
marker of a supervised learning problem is the inclusion of past observations and their target vectors.

2.1.1 Binary Classification

Smola and Vishwanathan define several other specific classes of problems, all of which have significant overlap with the broadly defined classes [17]. The most commonly encountered of those problem classes are Binary Classification, Multiclass classification, Regression, and Novelty Detection. Our research addresses the binary classification problem, which we describe herein.

Binary classification is perhaps the best-understood problem in machine learning [17]. Given a set of observations in a domain $X$ and their target values $Y$ as training data, determine the values $Y$ on the test data, where $Y$ is a binary value that classifies the observation. In general, the values of $Y$ are referred to as either positive or negative. This can be modified to suit the needs of the user. As an example, let us return to the problem of university admissions. A student who submits an application to a university will either be admitted or rejected. Although there may be other admission categories, such as "waitlist", for the purposes of this example we assume that admission or rejection are the only classes. In Table 2.1 we see 4 vectors of training data (GPA and SAT score) with their respective target vectors (Admission Status). The last vector is the test data we wish to classify.

<table>
<thead>
<tr>
<th>GPA</th>
<th>SAT Score</th>
<th>Admission Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>1850</td>
<td>Admit</td>
</tr>
<tr>
<td>2.9</td>
<td>2030</td>
<td>Reject</td>
</tr>
<tr>
<td>2.75</td>
<td>2180</td>
<td>Admit</td>
</tr>
<tr>
<td>3.33</td>
<td>1960</td>
<td>Reject</td>
</tr>
<tr>
<td>3.5</td>
<td>1710</td>
<td>? (Admit/Reject)</td>
</tr>
</tbody>
</table>

Table 2.1: Simple example of a classification problem
2.2 Model-Driven Engineering

The MDE paradigm relies on the usage of modeling languages that "formalize application structure, behavior, and requirements" \[18\] to drive the software engineering life-cycle. Often these models are paired with transformation engines that "analyze certain aspects of models and then synthesize various types of artifacts, such as source code, simulation inputs, XML deployment descriptions, or alternative model representations" \[18\]. The MDE paradigm grew out of earlier software abstraction efforts such as computer-aided software engineering (CASE), which arose in the 1980s and was very similar in concept to MDE but seldom used in practice \[18\]. Later efforts included the Object Management Group’s Model Driven Architecture (MDA), which was the direct precursor to MDE. In fact, MDE still incorporates the ideas of MDA, but goes one step further in adding both "process" and "analysis" to the "architecture" described by MDA \[10\]. In layman’s terms, one might think of MDE as the holistic incorporation of MDA into the software development life-cycle.

MDE is used in a wide variety of industries for many different applications \[19\]. Just as UML is language-agnostic, so too does MDE appear to be "problem-agnostic" in the sense that nearly any problem with a software solution can be addressed through MDE. It is a paradigm in the same manner as object-oriented programming (OOP) is a paradigm. And just as OOP describes the basic tenet that "everything is an object", MDE’s principle is that "everything is a model" \[20\].

Figure 2.1 \[20\] is a useful way of thinking about how these models are derived and structured. The model representing a software artifact would be at the $M_1$ level. To take a common example, the model for an automobile Engine Control Unit would be at this level. The $M_2$ level meta-model may be thought of as the language in which the model is expressed, hence the conformantTo descriptor connecting the $M_1$ and $M_2$ levels. Thus any language used for MDE, such as UML, would be found at level $M_2$. The $M_3$ level describes meta-meta-models, which are used to model the modeling languages themselves. Importantly, this meta-meta-model must conform to itself and in a sense is modeling itself. The Object Management Group defines a meta-meta-model standard called the Meta-Object Facility (MOF) from which many modeling languages are derived. The meta-model for UML is itself built from the MOF.
2.2.1 Domain-Specific Modeling Languages

Although UML is sufficient for modeling many programs, engineers might wish to develop their own domain-specific modeling languages (DSML) that are more tailored to their domain. DSMLs are often derived from UML, and there has been some research exploring standardization of this process [21]. This involves using the UML Profile mechanism to define novel relationships, with some work required in determining the closest match in semantics with existing UML artifacts. Although DSMLs are not always derived from UML, there are certain advantages in doing so. The primary benefit is allowing the reuse of extant UML tools. Note that these DSMLs are distinct from Domain-Specific Languages (DSL), which are usually programming languages in the standard sense. FORTRAN and COBOL can be thought of as examples of DSLs in the scientific and business domains [22].
2.2.2  Breuker’s Work - MDE For Big Data

To our knowledge and research, the only DSML designed to model machine learning was proposed by Breuker in 2014 [14]. This DSML was designed to bridge the gap between high demand for Big Data analysts and the lack of supply. Although designing and implementing algorithms for Big Data analytics is difficult and involved, there are several tools to facilitate the task. Breuker’s DSML represents a probabilistic graphical model (PGM). A PGM is a visual representation of a probabilistic model and is primarily comprised of variables and their relationships. Breuker uses Figure 2.2 as a simple example. Breuker defines requirements for PGMs to be represented as a DSML: 1) a "modeling language to express distributions as graphs" and 2) an inference algorithm that processes the graphs to "answer questions regarding conditional marginal distributions" [14]. Breuker’s DSML was built explicitly to work with the Infer.NET C# library. This library builds a PGM by having users define variables and connecting them with factors. The software compiles the inference model defined by the user’s code and runs the given inference algorithm. Infer.NET developers build the inference model through code rather than MDE-like, graphical, modeling. This is one significant shortcoming of the Infer.NET library and all other similar libraries.

Breuker’s DSML allows a user to build their inference models graphically rather than through code. We present their DSML in Figure 2.3. There is no accompanying defined code generation scheme, but Breuker does lay out a simple skeleton consisting of three methods within a single

![Figure 2.2: The probability distribution P(A,B,C) [14]](https://www.microsoft.com/en-us/research/project/infernet/)
C# class: *GenerateModel*, *InferPosteriors*, and *MakePredictions* [14]. In employing this DSML to help answer our research questions, we are also validating the DSML by using it in a full end-to-end realization.
Figure 2.3: Breuker’s DSML syntax [14]
2.3 Baseball Analytics

Due to its wealth of data and discrete nature, baseball readily lends itself to statistical analysis more than any other sport. Many books have been written on the subject, and in recent years baseball teams have embraced data-driven and statistical analysis prominently [23-25]. Most of the machine learning problem classes are applicable to baseball, although some have fewer practical uses than others [1, 2].

There are many forms of statistical analysis applied to baseball that do not relate to machine learning. Even simple statistics, such as batting average or a pitcher’s win-loss record, are often useful in determining success of a player or team. Prior to Bill James’s popularization of more complex analysis in the 1980s, these simple metrics served as the statistical foundation of baseball for decades [23].

One example of more complex analysis is Bill James’s Pythagorean expectation [26]. This is still a relatively simple formula, but it goes beyond the basic win-loss ratio to calculate the expected number of wins for a team given their runs scored and runs allowed. The formula is as follows:

$$\text{Expected Win Ratio} = \frac{\text{Runs Scored}^2}{(\text{Runs Scored}^2 + \text{Runs Allowed}^2)}$$

This Pythagorean expectation might be appropriate for a sports website or for amateur fans and analysts, as it is both simple to use and reasonably effective in its predictive power. The machine learning analyses such as the one we focus on in our research and the examples we present in the next section are more appropriate for professional analysts and academics interested in the field as they are both more complex and more powerful.

2.3.1 Machine Learning Applied to Baseball

Machine learning’s predictive power has led to its use in baseball for both practical and research applications [1, 2]. In general, analyses that employ machine learning improve with increasing numbers of observations [27]. This is more readily illustrated when one considers the extreme cases. Suppose the goal is to predict whether a pitcher’s next pitch will be a fastball or not. If
there are only one or two observations of the pitcher’s past pitches, it will be nearly impossible to predict the next pitch with any significant accuracy. However, if there are 100,000 observations, the accuracy of the prediction will be quite high. Due to baseball’s relatively large number of observations (162 games per season with 30 teams), machine learning is a very viable candidate for strong predictive power in baseball.

**Binary Classification**

Consider the example of predicting whether a pitcher’s next pitch will be a fastball or not. This is a binary classification problem in that there are two classes: fastball and non-fastball. Previous researchers have demonstrated excellent predictive improvements when using machine learning for this exact problem. Ganeshapillai and Guttag used a linear support vector machine classifier to classify pitches based on data from the 2008 season and predict the pitches of the 2009 season [28]. Support vector machines are a type of supervised learning that attempt to find a "separating hyperplane that leads to the maximum margin" [15]. The intuition behind this is to place the classes on different sides of a line or hyperplane, with a large margin extending on either side of the hyperplane. This is the type of learning that has feedback associated with it, such as labeled examples. In their experiments, the 2008 pitching data was the feedback, which contained labeled examples of the type of pitch that was thrown by the pitcher. The performance improvement witnessed by their approach over a naive classifier was approximately 18%. The naive classifier can be thought of as a simple Bayes classification based on probability. In other words, if a pitcher in 2008 used a fastball greater than 50% of the time, the naive classifier would predict that every pitch in 2009 would be a fastball. The model the two researchers created was able to correctly predict the next pitch 70% of the time, whereas the naive classifier was able to correctly predict the next pitch 52% of the time. This type of analysis using a support vector machine is one of the most widely used methods for binary classification.

There are nearly endless applications for binary classification in baseball. Simple contrived examples include classifying a matchup between two teams as a win or a loss, or classifying whether a certain pitcher against a certain player will choose to intentionally walk the player. In the former
example, the observations might consist of a vector of players and their individual performances using statistics such as on-base percentage or batting average. In the latter example, the training data might consist of the batter’s performance measured in batting average or slugging percentage. Actual examples addressing binary classification problems, and other classes, can be found in our survey paper on the subject [1][2]. For this research, we create a DSML and conforming model instances that address the binary classification problem of predicting if the next pitch will be a fastball.
Chapter 3

Approach

3.1 Overview

We present an overview of our approach in Figure 3.1. We begin by forming our metamodel in Papyrus, create model instances conforming to that metamodel, derive and test our code generation engine, and perform code execution and validation experiments.

![Figure 3.1: Our Approach](image)

3.2 Metamodel Formation

For our first task, we created a representation of the machine learning DSML in Papyrus [29]. Papyrus is an open-source software tool for supporting MDE. We chose to use Papyrus because it was free, open-source, and allows users to define and use their own DSMLs. We aimed to have the Papyrus metamodel match the metamodel specified by Breuker as closely as possible. We present
our representation in Figure 3.2. As Papyrus uses UML on the back-end, the basic elements of our DSML must inherit from the metaclass “Class.” Specifically the Node, Gate, GateOption, and Plate elements. The Variable and Factor elements both inherit from Node. Observed Variable and Random Variable both inherit from Variable. Barring the Papyrus-required and initial inheritance from the UML element “Class,” this is in accordance with the metamodel defined by Breuker.

Upon completion of this step of the MDE process, we enable modelers to build their own model instances that conform to and can be validated against the metamodel. Although we created a code generation scheme that is targeted at a limited binary classification use case only, our Papyrus metamodel will help future users target any use case and ensure that their model instance conforms to the metamodel. This is the first realization of the Breuker metamodel.

The crucial elements of this DSML are the Observed Variables, Random Variables, and Factors. To model a binary classification problem and take advantage of our code generation engine, a modeler implementing an instance model first designates the training data features as Observed Variables. They then choose a Variable that will be predicted with test data. This “predict” Variable should be connected to the features by a Factor, which contains a Random Variable representing a weight matrix. Any number of features corresponding to Observed Variables can be modeled, but only one Factor/Random Variable pair may be used. In addition, there should be only one Observed Variable to be predicted.
Figure 3.2: The metamodel in Papyrus
3.3 Creation of Different Model Instances

To help validate our DSML metamodel and answer our research question regarding model updates/instances, we decided to create several variations of model instances conforming to the metamodel. Each of these model instances must be capable of predicting if the next pitch will be a fastball and vary by using different features for pitch classification. For example, Figure 3.3 uses count as the sole predictive factor for the next pitch. A refined version of this model instance we created, and present in Figure 3.4, uses the count and the pitcher’s Earned Run Average (ERA) as the factors to predict the pitch. For these two simple examples, we use Papyrus’ built-in model validation tool to validate these model instances, finding no errors or warnings, indicating that they conform to the metamodel.

These basic model instances were designed to represent a feasible real-world application of baseball analytics. The use of “count” and “ERA” as predictive features is common when evaluating a pitcher’s behavior. While these may be simple examples, they are appropriate for an amateur/novice analyst. These examples also help us confirm that Papyrus’ model validation tool is interpreting our metamodel correctly as we can easily cross-validate these models by hand. We also built a model instance that should not conform to the metamodel, wherein a Factor is missing its Random Variable. We present this in Figure 3.5, the red “X” on the Factor indicates that this is the cause of the model’s invalidity.

The final model instance that we built consisted of 18 factors adapted from the work of Ganeshapillai and Guttag [28]. Due to our inability to obtain the original data source, we were limited in our ability to model the original 36 features. These factors are Inning, Handedness, NumberOfPitches, Strikes, Balls, Outs, BasesLoaded, HomePrior, CountPrior, BattingTeamPrior, BatterPrior, HomePriorSupport, CountPriorSupport, BatTeamPriorSupport, BatterPriorSupport, PreviousPitchType, PreviousPitchResult, and PreviousPitchVelocity. We further define these factors/terms in our glossary. We present the full model instance in Figure 3.6. Because we are using so many factors, it’s difficult to build such a model and have it look aesthetically pleasing. In the next section, we discuss the code generation engine we designed that automatically generates code from these model instances.
Figure 3.3: Basic Model Instance with Count as Sole Factor (Strikes and Balls)
Figure 3.4: Less Basic Model Instance with Count and Pitchers ERA as factors
Figure 3.5: Invalid Model Instance - Factor Missing its Random Variable
Figure 3.6: Complete Model Instance with 18 Factors
3.4 Devising the Code Generation Engine

In this step, we had to devise a code generation engine that 1) parses models conforming to our DSML and 2) generates a C# file that can be used out-of-the-box to make predictions on test data. We completed this step nearly from scratch as Breuker provided no code example, instead merely proposing a code layout. Our goal was to have the resulting C# code use the Infer.NET library to create a coded abstraction of the user-defined model. We wrote our code template using EGX, EGL, and the Epsilon Object Language (EOL) [30], which were designed for MDE.

Although we represented the entire DSML as a metamodel in Papyrus that can be used for constructing models, the Factors and Observed & Random Variables are the only ones that we accounted for in our code generation engine. We ignored the other constructs, such as Gate, GateOption, and Plate as they were unnecessary for our case study to demonstrate plausibility of using MDE to create machine learning software without writing code manually.

We present our generated code template in Listing 3.1. In the Epsilon family of languages, the text in the code template that is enclosed in "[\%]" brackets is dynamic and variable based on user input. All other text in the template is static. Our first step in parsing is to have the engine collect the Observed Variables and determine which of them is to be predicted by the generated software. We then have the engine create an array for each Observed Variable containing its training values and insert it into the code. This step also inserts the Boolean array of training values for the Observed Variable that will be predicted by the software. The generated code proceeds to initialize a weight matrix with random numbers from a Gaussian distribution, with the size of the matrix corresponding to the number of features.

This training data is passed by the generated code to a method that updates the weights based on the training data, using the Expectation Propagation algorithm [31] to infer the posterior distribution of the weight matrix. In machine learning, this is referred to as training the model. The generated code then creates arrays to hold the test data and makes predictions on this data. The output consists of a probability for each observation, indicating the likelihood that the given observation belongs to the “true” class. Recall that this is due to the target training data consisting of a Boolean “true/false” array. In our baseball analytics use case, the output indicates the probability of the next pitch being
a fastball.

Listing 3.1: The code generation engine

```csharp
[%
var findObvStereotype = Stereotype . all . selectOne ( s | s . name = "Observed/uni2423Variable ");
var findRandomStereotype = Stereotype . all . selectOne ( s | s . name = "Random/uni2423Variable ");
var predVar = Class . all ;
var classes = Class . all ;
var observedVarCount = 0;
var nonPredObserveds = new Bag;
for (c in classes) {
    if (c . getAppliedStereotypes . includes ( findObvStereotype )) {
        observedVarCount++;
        if (c . stereotypeApplications . predict . first ()) {
            predVar = c ;
        } else {
            nonPredObserveds . add(c);
        }
    }
}%

using System ;
using System . Collections . Generic ;
using System . Text ;
using MicrosoftResearch . Infer . Models ;
using MicrosoftResearch . Infer ;
using MicrosoftResearch . Infer . Distributions ;
using MicrosoftResearch . Infer . Maths ;
namespace ML_DSML {
    public class DSMLCodeGen {
        public static void Main( string[] args ) {
            // The labeled training data
            [%
            for (c in classes) {
                if (c . getAppliedStereotypes . includes ( findObvStereotype )) {
                    if (c . stereotypeApplications . dataType . first () == "Double") { %]
                        double[] [%=c . name%] = [%=c . stereotypeApplications . trainValues . first ()%];
                    } else {%
                        bool[] [%=c . name%] = [%=c . stereotypeApplications . trainValues . first ()%];
                    }
            }
        }%
        // Create target vector and weights
        VariableArray<bool> y = Variable . Observed([%=predVar . name%]).Named("y");
        Variable<Vector> w = Variable . Random(new VectorGaussian ( Vector . Zero([%=observedVarCount%]),
            PositiveDefiniteMatrix . Identity ([%=observedVarCount%])).Named("w");
        BayesPointMachine([% for (o in nonPredObserveds) [%][%o . name%], [%][%w, y];
    } // for inference engine object and infer posterior distribution
// of weights
```
InferenceEngine engine = new InferenceEngine();
VectorGaussian wPosterior = engine.Infer<VectorGaussian>(w);
Console.WriteLine(" Dist/uni\over/uni\w = " + wPosterior);

// make predictions on test data
double[] [o in nonPredObserveds] [\%o.name\%]Test = [\%o.stereotypeApplications.testValues.first()\%];
VariableArray<bool> ytest = Variable.Array<bool>(new Range([%=nonPredObserveds.first().name%]Test.Length)).Named("ytest");
BayesPointMachine([% for (o in nonPredObserveds) %]([%=o.name%]Test, [%]Variable.Random(wPosterior).Named("w"), ytest);
Console.WriteLine(" output = " + engine.Infer(ytest));

public static void BayesPointMachine([% for (o in nonPredObserveds) %]double[] [%=o.name%], [%]Variable<Vector> w, VariableArray<bool> y)
{
    // Create training data vector with bias parameter of 1
    Range j = y.Range;
    Vector[] xVector = new Vector[[%=nonPredObserveds.first().name%].Length];
    for (int i = 0; i < xVector.Length; i++)
        xVector[i] = Vector.FromArray([% for (o in nonPredObserveds) %]1, [%]1);
    VariableArray<Vector> x = Variable.Observed(xVector, j).Named("x");

    // Bayes Point Machine, dot product of weights and feature vector
    double noise = 0.1;
    y[j] = Variable.GaussianFromMeanAndVariance(Variable.InnerProduct(w, x[j]).Named("innerProduct"), noise) > 0;
}

In Listing 3.2 we present the code generated from our full model using a small artificial data set for demonstration purposes. We discuss real data later on.

Listing 3.2: Example of automatically generated code

In Listing 3.2, we present the code generated from our full model using a small artificial data set for demonstration purposes. We discuss real data later on.
double[] handedness = {1, 1, 1, 1, 1, 0};
double[] numPitches = {18, 32, 9, 1, 88, 34};
double[] homePrior = {.25, .36, .88, .75, .99, .92};
double[] battingTeamPrior = {.15, .78, .59, .75, .14, .06};
double[] countPrior = {.89, .18, .72, .72, .1, .49};
double[] batterPrior = {.7, .53, .91, .56, .9, .8};
double[] homePriorSupport = {180, 58, 75, 234, 75, 250};
double[] battingTeamPriorSupport = {44, 58, 59, 204, 167, 61};
double[] countPriorSupport = {32, 237, 151, 56, 58, 131};
double[] batterPriorSupport = {193, 226, 244, 9, 153, 122};
double[] batterSlugging = {.182, .493, .337, .155, .157, .372};
double[] batterRuns = {37, 84, 12, 35, 53, 74};
double[] previousPitchType = {1, 0, 1, 1, 0, 1};
double[] previousPitchResult = {1, 0, 0, 1, 1, 0};
double[] previousPitchVelocity = {87, 85, 89, 91, 97, 101};
double[] previousPitchVelocityGradient = {12, 1, -13, 1, 13, 12};

// Create target vector and weights
VariableArray<bool> y = Variable.Observed(fastball).Named("y");
Variable<Vector> w = Variable.Random(new VectorGaussian(Vector.Zero(23),
PositiveDefiniteMatrix.Identity(23))).Named("w");
BayesPointMachine(strikes, balls, outs, scoreDifferential, basesLoaded, inning, handedness, numPitches, homePrior, battingTeamPrior,
countPrior, batterPrior, homePriorSupport, battingTeamPriorSupport, countPriorSupport, batterPriorSupport,
batterSlugging, batterRuns, previousPitchType, previousPitchResult, previousPitchVelocity, previousPitchVelocityGradient,
w, y);

// Create inference engine object and infer posterior distribution
// of weights
InferenceEngine engine = new InferenceEngine();
VectorGaussian wPosterior = engine.Infer<VectorGaussian>(w);
Console.WriteLine(" Dist/uni2423over/uni2423w=
" + wPosterior);

// create inference engine object and infer posterior distribution
// of weights
InferenceEngine engine = new InferenceEngine();
VectorGaussian wPosterior = engine.Infer<VectorGaussian>(w);

// make predictions on test data
double[] strikesTest = {2, 1, 2};
double[] outsTest = {3, 2, 0};
double[] scoreDifferentialTest = {1, 0, 1, 2, 1, 3};
double[] basesLoadedTest = {0, 0, 0, 1, 0, 0};
double[] inningTest = {3, 1, 3, 2, 9, 6};
double[] handednessTest = {0, 1, 1, 0, 1, 1};
double[] numPitchesTest = {36, 20, 29, 11, 68, 5};
double[] homePriorTest = {.05, .87, .77, .62, .86, .70};
double[] battingTeamPriorTest = {.4, .45, .58, .02, 1, .7};
double[] countPriorTest = {.03, .38, .92, .06, .7, .23};
double[] batterPriorTest = {.08, .1, .04, .59, .90, .11};
double[] homePriorSupportTest = {10, 115, 104, 166, 171, 173};
double[] battingTeamPriorSupportTest = {162, 151, 27, 207, 144, 163};
double[] countPriorSupportTest = {38, 22, 114, 130, 195, 182};
double[] batterSupportTest = {140, 190, 57, 233, 194, 9};
double[] batterSluggingTest = {.237, .172, .222, 2, .393, .282};
double[] batterRunsTest = {24, 48, 90, 54, 89, 75};
double[] previousPitchTypeTest = {1, 1, 1, 0, 1, 1};
double[] previousPitchResultTest = {1, 0, 1, 0, 1, 0};
double[] previousPitchVelocityTest = {94, 88, 84, 101, 101, 110};
double[] previousPitchVelocityGradientTest = {-5, -11, -2, -9, 1, 1};
VariableArray<bool> yTest = Variable.Array<bool>(new Range(strikesTest.Length)).Named("ytest");
BayesPointMachine(strikesTest, outsTest, scoreDifferentialTest, basesLoadedTest, inningTest, handednessTest,
numPitchesTest, homePriorTest, battingTeamPriorTest, countPriorTest, batterPriorTest, homePriorSupportTest, battingTeamPriorSupportTest, countPriorSupportTest, batterPriorSupportTest, batterSluggingTest, batterRunsTest, previousPitchTypeTest, previousPitchResultTest, previousPitchVelocityTest, previousPitchVelocityGradientTest, Variable.Random(wPosterior).Named("w"), ytest);

Console.WriteLine("output=ln" + engine.Infer(ytest));
}

public static void BayesPointMachine(double[] strikes, double[] balls, double[] outs, double[] scoreDifferential,
  double[] basesLoaded, double[] inning, double[] handedness, double[] numPitches, double[] homePrior, double[] battingTeamPrior, double[] countPrior, double[] batterPrior, double[] homePriorSupport, double[] battingTeamPriorSupport, double[] countPriorSupport, double[] batterPriorSupport, double[] batterSlugging, double[] batterRuns, double[] previousPitchType, double[] previousPitchResult, double[] previousPitchVelocity, double[] previousPitchVelocityGradient, Variable<Vector> w, VariableArray<bool> y)
{
  // Create training data vector with bias parameter of 1
  Range j = y.Range;
  Vector[] xVector = new Vector[strikes.Length];
  for (int i = 0; i < xVector.Length; i++)
    xVector[i] = VectorFromArray(strikes[i], balls[i], outs[i], scoreDifferential[i], basesLoaded[i], inning[i], handedness[i], numPitches[i], homePrior[i], battingTeamPrior[i], countPrior[i], batterPrior[i], homePriorSupport[i], battingTeamPriorSupport[i], countPriorSupport[i], batterPriorSupport[i], batterSlugging[i], batterRuns[i], previousPitchType[i], previousPitchResult[i], previousPitchVelocity[i], previousPitchVelocityGradient[i], 1);
  VariableArray<Vector> x = Variable.Observed(xVector, j).Named("x");

  // Bayes Point Machine, dot product of weights and feature vector
  double noise = 0.1;
  y[j] = Variable.GaussianFromMeanAndVariance(Variable.InnerProduct(w, x[j]).Named("innerProduct"), noise) > 0;
}

3.4.1 Quality of Generated Code

To help assess the quality of the generated code, we enlisted the help of an undergraduate student, Kelsea McGraw. As part of their undergraduate honors project in a Software Testing course, they performed quality assessment and criteria-based testing on our generated code. Specifically, they used input space partitioning to test the code by devising several different inputs as training and test data. For instance, using valid integers for the "strikes" data that do not make sense in the context of baseball, such as a training sample where the "strikes" observation is equal to 10. Overall, they found the quality of the code to be fairly high, with some important areas for improvement. The generated code can still make predictions given nonsensical data, but the value of these observations is limited. We plan on addressing such concerns in future work by placing limits on the possible
values that can be entered in Papyrus.

### 3.5 Code Execution and Validation

The actual data set we used consisted of the play-by-play pitching statistics from the 2016 and 2017 MLB seasons of the Cincinnati Reds, New York Yankees, New York Mets, and Toronto Blue Jays. Specifically, we considered all pitchers from these four teams in 2016 and used their 2017 pitch data to test our prediction model. Even if a pitcher switched teams in 2017, we still considered their pitch data on that new team. Similarly, we also removed pitchers from the 2016 data who retired or were free agents during the 2017 season. We retrieved this data from Baseball Savant, which hosts official Major League Baseball data. Through this interface, we were able to enter manual queries that gave us play-by-play data for these 4 teams/pitchers in 2016 and 2017. Each season consisted of roughly 85,000 observations. We have uploaded this training and test data to our repository. We used the 2016 season data as training data and the 2017 season as the test data to evaluate our prediction accuracy.

After building our final model instance, we needed a way to feed it our large data set of approximately 85,000 observations. Papyrus allows simple entry through its interface, but no systematic method of inputting large strings of information. To feed data to the model, we had to enter the observations for each Observed Variable as a string in the format \{x-1, x-2, ..., x-n\}, where \(n = \text{NumberOfObservations}\). Thus, we first used Python and the Pandas library to read our data which was organized as a comma separated values (CSV) file. We then extracted relevant columns before feeding them to the function shown in Listing 3.3, which takes in a Pandas Series and outputs the appropriately formatted series. This was very helpful for us in formatting the data in this manner and hopefully can be of future use to Papyrus users, who can download it on our repository.

Listing 3.3: Python function to output properly formatted string

```python
def to_string(series):
```

1. https://baseballsavant.mlb.com
2. https://pandas.pydata.org/
# Takes in a Pandas series

```python
string = '{
for index, item in series.iteritems():
    if index == series.shape[0] - 1:
        string += str(item) + '}
    else:
        string += str(item) + ',',

return string
```

To help validate our model and the process as a whole we fed this formatted data to our model instance in Papyrus, subsequently generating an executable C# file. After running the C# file, we obtained a list of output probabilities for each test observation. We then imported these probabilities into Python as a Pandas Series and classified them as 'true' if the probability was greater than 50%, otherwise they were classified as false. This was then compared against our actual test data from 2017. We further compared our predictions and the predictions that would be made by a naive classifier. The naive classifier classifies the 2017 observations based on the pitcher’s overall prior probability from 2016. In other words, if the pitcher in 2016 threw fastballs more than 50% of the time, the naive classifier would classify all pitches in 2017 as fastballs. We discuss these results and the subsequent prediction accuracies in Section 4.
Chapter 4

Results

In this section, we present the quantitative results of using our DSML full-factored model instance and automatic code generation to approach this baseball analytics binary classification problem. After formatting our data, building the model, feeding the data into the model, and running our automatically generated software, our software exhibited a prediction accuracy of 71.36%. That is to say that, our prediction model was able to successfully determine if the next pitch was a fastball or not 71.36% of the time in the 2017 season. This is slightly better than the results obtained by Ganeshapillai and Guttag, as their prediction accuracy was roughly 70% \[28\], albeit on a larger data set and different year as we address in our discussion.

While our prediction accuracy is higher than Ganeshapillai and Guttag, we must note that our model exhibited a lower increase over the naive classifier than their work. The naive classifier predicts pitches based on the overall prior probability of that pitcher throwing a fastball. For example, if a pitcher in 2016 threw fastballs at a rate of 51%, the naive classifier would simply classify all pitches in 2017 as a fastball because it is always more likely that the pitcher will throw a fastball than not. When using a naive classifier for our data set, there was a prediction accuracy of 61.72%. Thus, we achieved about 15.6% improvement against the naive classifier. This is certainly a positive result as we have demonstrated a significant improvement in accuracy. However, Ganeshapillai and Guttag achieved roughly 18% improvement in prediction accuracy, which is somewhat higher than ours. We consider this further in our discussion, however, our goal
was to validate the plausibility of the MDE approach to building machine learning software. The key quantitative takeaway is that our automatically generated C# code gave us an improvement in prediction accuracy over a naive classifier. Figure 4.1 illustrates the prediction accuracy of our work versus that of Ganeshapillai and Guttag.
Figure 4.1: Prediction Accuracy for our Work and Ganeshapillai and Guttag’s [28]
Chapter 5

Discussion

5.1 Research Questions

At the beginning of this project, we set out to answer two main research questions. For our first question, we were able to successfully encapsulate machine learning binary classification concepts in a DSML and an accompanying code generation scheme. Although we demonstrated this with a baseball analytics use case, our DSML will allow a user to enter whatever data they wish, as long as there are training and test observations. We successfully built a code generation engine that parses a model to build machine learning software and is fairly robust with respect to both number of features and number of observations. For the second part of our first research question, we encountered several challenges which we document in Section 5.3.

For our second research question, we realized a complete application of MDE for machine learning that allows for model updates and code generation through a baseball analytics use case. We created several model instances to demonstrate the ease of model updates. We essentially built our model instances in a step-by-step incremental fashion. We facilitated generated code for a large and fully-featured predictive model that can be used for meaningful analysis. Through the use of MDE software models, we automatically generated executable code, which is the essence of MDE.
5.2 Threats to Validity

Most prominently, our training and test data is different than the set that Ganeshapillai and Guttag \cite{28} used for their paper. We used the 2016 MLB season as training data and the 2017 season as test data, whereas they used the 2008 MLB season as training data and the 2009 season as test data. We reached out to both researchers and the owners of the data and they were unwilling to provide us access to that data. The data was available for purchase but at a prohibitively expensive cost. Since we were more concerned with the plausibility of using MDE for this binary classification problem and only exhibiting comparable results to the traditional approach, we decided having their exact data was unnecessary and that real-life baseball data, albeit from a different year, was acceptable. It is still a threat to validity however. It may be the case that had we used their older data set, that our prediction accuracy increase would be quite different. The game of baseball has changed significantly since the time of that data. Baseball teams have invested heavily in statistical analysis departments and consequently changed their approach to pitching.

We used a smaller set of factors than Ganeshapillai and Guttag did in their work. As a result of being unable to obtain the original curated data set, we had to resort to the limited data set provided by the free Baseball Savant website. Importantly, we did not have access to play-by-play statistics for the batter that was going up against the pitcher. This inability to build a deep batter profile likely resulted in a loss in our prediction accuracy. Although the batter was taken into account, it was only through simple identification and pitcher-batter priors. Due to our necessary onerous manual web querying, we decided to use 4 MLB teams as our baseline rather than the entire league. We used the New York Mets, New York Yankees, Toronto Blue Jays, and Cincinnati Reds as these were teams of interest to us. Undoubtedly, this is a limitation and threat to validity to our prediction accuracy. Despite this limitation, our work still shows the feasibility of building machine learning software through the MDE paradigm, and our prediction accuracy was very similar to that of Ganeshapillai and Guttag. Achieving this without having to write any source code manually is still a significant result.

Another important threat to validity is our use of a different algorithm for making predictions. Ganeshapillai and Guttag use a Support Vector Machine to classify their pitches. Because we were
extending a DSML that was intended for use with the Infer.NET library, we were limited in the algorithms that were available to us. In particular, Infer.NET has no constructs that would allow for a Support Vector Machine. Because the Infer.NET library is based on inference learning for probabilistic graphical models, the algorithms are likewise limited. We had to use a Bayes Point Machine classifier, which may have impacted potential prediction accuracy.

Finally, our metamodel and code generation engine were built to work with Papyrus. We chose to use Papyrus as it is free and open-source, can work on all major operating systems (Linux, macOS, Windows), and has growing support in both research and industry. This is a threat to validity as we did not consider other MDE DSML tools. To simplify future experiments in Papyrus and other tools, we’ve uploaded a Papyrus project that can be imported directly into Papyrus or converted for us in other tools [32].

5.3 Lessons Learned and Challenges

Part of our goal in research was to help identify lessons that we learned and interesting challenges. We faced several challenges throughout the course of this project.

Before beginning the research and modeling process, we first needed to gather real-world Major League Baseball data, ideally the same data as that used by Ganeshapillai and Guttag [28]. Upon contacting their data source, STATS Inc., we were told that such data was no longer publicly available and the quoted purchase price was prohibitively expensive. As a result, we decided to use Baseball Savant’s web interface. This website did not have the 2008 nor 2009 data, thus we decided to use the more recent 2016 and 2017 data. While this data was sufficient for our goals and research questions, this experience calls attention to one of the major problems in the machine learning field of finding relevant and clean data. Future researchers should keep this in mind as a high priority for their projects.

Another challenge we faced was the learning curve in using EGX and EGL to write the code generation engine. We found the documentation for both of these somewhat lacking. In the process of learning how to use these components of the Epsilon Object Language, we had to make multiple forum posts on the Epsilon forum. This is fair and to be expected for an open-source language.
Like our previous issue, future researchers who wish to use EGX and EGL should be aware of the time investment required to sufficiently comprehend the API. Searching and contributing forum posts on the official Epsilon forum may prove useful, as it did for us.

The open-source nature of both Papyrus and the Epsilon Object Language was probably the most significant challenge we encountered, albeit a tool-related one. In addition to having to rely on the open-source community, we found ourselves wanting greater customization options for the Papyrus dashboard. In particular, when a user uses our metamodel to instantiate their own model instance, they must first create it as a UML class. They must then click on this class and successively follow the Properties > Profile > Applied Stereotypes menu chain before selecting their desired stereotype. These stereotypes correspond to our DSML constructs of Observed Variable, Random Variable, and Factor. This somewhat onerous process significantly impacts user workflow. Another obstacle we faced in the using Papyrus is the method required to feed data to the model. As discussed in Section 3.5 there is no systematic way to feed large strings of information to the model, so it must be done manually. This becomes impractical for users to enter manually when working with any non-trivial data set.

5.4 Potential Impact and Future Work

The target audience for this work is domain experts who want to build machine learning software without manually writing source code. This can include organizations that possess the requisite machine learning and computer science expertise, but cannot spare the time or resources to write code in a traditional manner. Although our work’s scope is limited in that it applies to binary classification problems only, it is fairly robust in regards to its number of features and number of training observations.

An important benefit of this work is that our code generation engine, models, and other artifacts have been posted and are open-source and free to use. This makes it a cost-effective solution, or starting point, for individuals or organizations that are building binary classification systems. Even if individuals are not interested in our code generation engine, we believe that our metamodel representation in Papyrus can serve as a valuable tool for future practitioners building probabilistic
graphical models.

We anticipate this work will help serve as a stepping stone to future research by moving towards proving the viability of MDE in the machine learning domain. The industrial use case of baseball analytics is an entirely relevant and economically viable one to demonstrate the potential impact of this work. Future researchers, academic or industrial, can build off of this project to further flesh out the code generation engine and make it robust for other machine learning problem classes, and potentially address the Gate, GateOption, and Plate constructs that we deemed beyond scope. All of our artifacts have been posted for open use on our repository [32].

An interesting area of future work would be to assess the degree of difficulty involved in using our approach. For example, we could recruit volunteers to help further support our claim that MDE is viable in the machine learning domain. We deemed this beyond the scope of the project at this time.
Chapter 6

Related Work

To our knowledge, there is little other related work in creating a Machine Learning DSML. The paper by Breuker [14] was only exploratory in nature, and has not been cited in anything other than survey papers until this time. Ours appears to be the first original work in realizing this DSML and demonstrating its viability for building quality machine learning software.

There are several software packages which perform similar functions in rapidly applying machine learning algorithms on sets of data. One example is the Orange package, which is part of the Anaconda Python distribution. Orange can be more thought of, and is advertised as, a data mining suite [33]. The crucial difference between our work and Orange is that Orange does not adhere to nor support the MDE approach to software engineering. Although users interact with a visual interface and can connect certain components like "Data" or "Analysis" to one another, there is no model validation and no resultant automatically generated software. Rather, the Orange package allows one to apply a machine learning algorithm to user fed-in data, assess prediction accuracy, and build visualizations.

WEKA is a similar package to Orange, which allows users to perform data mining on their data sets with a variety of different machine learning algorithms [34]. WEKA bears even less of a resemblance to the MDE paradigm than Orange. There is no visual connecting of components like there is with Orange. Consequently, WEKA does not allow a user to define a model for what their program should look like.
TensorFlow is a machine learning framework developed by Google Brain that is used in training neural networks. The idea behind TensorFlow is very similar to that of Infer.NET, in that the user defines models of behavior using code. While Infer.NET is used for defining probabilistic graphical models, TensorFlow is used primarily to define neural network architectures. Although TensorFlow does not define itself as a Model-Driven Engineering language in any traditional sense, one can argue a neural network architecture defined in TensorFlow is analogous to a software engineering model. A significant difference in their work and ours is TensorFlow does not allow the user to define/create neural network architectures in a visual manner. The Tensorboard feature allows the user to view existing models that have already been built through user-defined code. Some potential research we may consider includes developing some type of visual modeling layer on top of TensorFlow, very similar to what we have accomplished with this work, allowing the user to define TensorFlow models without writing any code by hand.
Chapter 7

Conclusion

In this thesis, we considered the plausibility of using Model-Driven Engineering to build machine learning software. Much like Breuker [14], we were motivated by the high demand for machine learning expertise and the shortage of individuals with the technical knowledge required to build such software systems. In particular, there are many domain experts in various fields who could benefit from such software but have neither the expertise nor resources required. We hoped that MDE could help address this shortfall, and thus looked for other research that had been performed in this domain. Breuker’s incomplete proposal for a machine learning DSML was the only one we found. Although incomplete, the metamodel they proposed was a direct mapping of the constructs in the Infer.NET library, which are themselves constructs taken from probabilistic graphical models.

We defined Breuker’s metamodel in Papyrus, an open-source package that allows users to describe their own modeling languages and create model instances conforming to that language. Thus, our work allows a user to build a probabilistic graphical model in Papyrus and have the native model validation tool ensure that the model instance is in accordance with the metamodel. To confirm this, we built several model instances of a baseball analytics problem: classifying pitches as fastball/non-fastball based on past data. The simple models allowed us to determine the ease of quickly building and defining our software through models. It is relatively trivial, for example, to update a model predicting fastballs based on "strikes" and "balls" to also take into account the "inning" at the time of the pitch.
Further, we built a code generation engine using EGX, EGL, and the Epsilon Object Language that would allow the user to build models to solve binary classification problems. Our use case took 2016 MLB data from pitchers for four teams as training data and 2017 data from those same pitchers as test data. Building a model that emulated many of the features from work by Ganeshapillai and Gutttag [28], we successfully achieved a comparable increase in prediction accuracy over a naive classifier. Most importantly, this was done through automatically generated code that we used to make predictions. Based on this finding, we have demonstrated the plausibility of a complete application of machine learning MDE, and proven the viability of MDE in the machine learning domain.
Glossary

balls  A variable indicating the number of balls when pitch is thrown.

basesLoaded  A boolean variable indicating if all three bases are loaded (a player is on them) when pitch is thrown.

batterPrior  A variable indicating the percentage of fastballs thrown when the pitcher is facing the current batter.

batterPriorSupport  A variable indicating the number of observations for the batterPrior variable.

battingTeamPrior  A variable indicating the percentage of fastballs thrown when the pitcher is facing the current team.

battingTeamPriorSupport  A variable indicating the number of observations for the battingTeamPrior variable.

countPrior  A variable indicating the percentage of fastballs thrown when the pitcher is at the current count (strikes and balls).

countPriorSupport  A variable indicating the number of observations for the countPrior variable.

fastball  A boolean variable indicating if the pitch was a fastball (true) or not (false).

handedness  A boolean variable indicating if the pitcher and batter have the same handedness.

homePrior  A variable indicating the percentage of fastballs thrown when the pitcher is playing at home.
**homePriorSupport**  A variable indicating the number of observations for the homePrior variable.

**inning**  A variable indicating the current inning when pitch is thrown.

**numPitches**  A variable indicating the number of pitches thrown by the pitcher up to the current pitch in game.

**outs**  A variable indicating the number of outs when pitch is thrown.

**previousPitchResult**  A variable indicating the result of the pitch thrown by the pitcher prior to current pitch. Was it a hit, a strike, a ball, etc.

**previousPitchType**  A boolean variable indicating the type of pitch thrown by the pitcher prior to current pitch. True if fastball, false if not.

**previousPitchVelocity**  A variable indicating the speed of the pitch thrown by the pitcher prior to current pitch.

**strikes**  A variable indicating the number of strikes when pitch is thrown.
Bibliography


