Realization of a Machine Learning Domain Specific Modeling Language: A Baseball Analytics Case Study

Kaan Koseler, Kelsea McGraw and Matthew Stephan
Dept. of Computer Science and Software Engineering, Miami University, 510 East High Street, Oxford Ohio, U.S.A.

Keywords: Machine Learning, Domain Specific Modeling Language, Baseball Analytics, Binary Classification, Supervised Learning, Model Driven Engineering.

Abstract: Accompanying the Big Data (BD) paradigm is a resurgence in machine learning (ML). Using ML techniques to work with BD is a complex task, requiring specialized knowledge of the problem space, domain specific concepts, and appropriate ML approaches. However, specialists who possess that knowledge and programming ability are difficult to find and expensive to train. Model-Driven Engineering (MDE) allows developers to implement quality software through modeling using high-level domain specific concepts. In this research, we attempt to fill the gap between MDE and the industrial need for development of ML software by demonstrating the plausibility of applying MDE to BD. Specifically, we apply MDE to the setting of the thriving industry of professional baseball analytics. Our case study involves developing an MDE solution for the binary classification problem of predicting if a baseball pitch will be a fastball. We employ and refine an existing, but untested, ML Domain-Specific Modeling Language (DSML); devise model instances representing prediction features; create a code generation scheme; and evaluate our solution. We show our MDE solution is comparable to the one developed through traditional programming, distribute all our artifacts for public use and extension, and discuss the impact of our work and lessons we learned.

1 INTRODUCTION

Using data to derive patterns is a critical driver of human knowledge and progress (Mattson, 2014). Due to the global proliferation and ubiquity of information processing devices, we have more data about the world than ever before. "Big Data" is a popular term that captures the essence of this era, representing the idea of there being more data than human beings can process without the assistance of computers and algorithms. The individuals or organizations that can leverage this data to derive insights are richly rewarded (Zimmerman, 2012). However, this almost always requires the deployment of advanced data processing and analysis techniques. One particularly popular technique is machine learning. A generally accepted definition of machine learning is finding patterns in data that "provide insight or enable fast and accurate decision making" (Witten et al., 2016). This usually takes the form of an output of predictions on new examples.

The explosive growth of machine learning has introduced several new problems and challenges for industries. Due to the esoteric nature of machine learning, it is difficult to find software engineers that posses mastery of machine learning techniques and domain knowledge, or data analysts with software engineering abilities (DeLine, 2015). One possible approach to address this industry problem is to utilize Model-Driven Engineering (MDE), which is a paradigm focused on formal abstractions to build a "model" of a particular application or software artifact (Kent, 2002). 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, and can include automatic code generation through interpretation of that 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. Additionally, with domain-specific modeling languages (DSML) (Kelly and Tolvanen, 2008), engineers can create models using a syntax consistent with terms and abstractions from their domain.

For this paper, we attempt to help address this industrial need by applying MDE using a machine

Koseler, K., McGraw, K. and Stephan, M.
Realization of a Machine Learning Domain Specific Modeling Language: A Baseball Analytics Case Study.
DOI: 10.5220/0007245800130024
In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development (MODELSWARD 2019)*, pages 13-24
ISBN: 978-989-758-358-2

learning DSML to an industrial setting and data. Specifically, we conduct a case study involving the industry of professional baseball analytics. Baseball is particularly suited for this type of analysis due to its rigidly discrete nature and wealth of statistical information over its long history. Machine learning techniques are increasingly being incorporated by organizations into the baseball industry (Sawchik, 2015). It is a multi-billion dollar, competitive industry that has wholeheartedly embraced data-driven analytics, and thus presents an excellent case study for us to assess the feasibility of applying MDE in a popular and practical industry. Additionally, the scale of the data provided by professional baseball is considerable. At the highest levels of North American competition, teams play 162 games per year that each generate thousands of data points.

Previous work by Breuker defined a syntax for a DSML that abstracts machine learning techniques (Breuker, 2014). However, it was never applied, implemented, nor validated. Our project, which is based on our thesis research (Koseler, 2018), serves as the first-time realization of their syntax and language in a popular and appropriate domain (Koseler and Stephan, 2017b), including a definition of a code generation scheme for a baseball binary classification problem. While our model instances are specific to our baseball case study, the DSML itself is not specific to baseball. We have published our refined DSML, model instances, code generation scheme, data cleaning scripts, and more artifacts on our repository to help enable future use of modeling in machine learning applications and reproduction of our study [1]. We set out to answer the following questions in our case study,

1. Is it possible to encapsulate machine learning binary classification concepts in a DSML and accompanying code generation scheme?
   (a) What lessons did we learn in doing so?
2. Can we realize a complete application of machine learning MDE in the context of a Big Data baseball use case that allows for various model instances and code generation?

We begin in Section 2 by giving a brief overview of machine learning, baseball analytics, and Breuker's work (Breuker, 2014). In Section 3, we describe our method towards demonstrating the feasibility of MDE in the machine learning domain. In Section 4, we present and discuss the results of our method, particularly our prediction accuracy. We additionally address our research questions, explore potential threats to validity, explain the challenges we faced along with the corresponding lessons we learned, and address the practical impact of our work and how it may be expanded upon in the future. In Section 5, we present related work and in Section 6 we present our conclusion.

[1] https://sc.lib.miamioh.edu/handle/2374.MIA/6234

2 BACKGROUND

In Section 2.1, we introduce general concepts of supervised machine learning and the binary classification problem, which are the focus of our case study. In Section 2.2, we overview the field of baseball analytics and its intersection with machine learning. In Section 2.3, we present Breuker's preliminary work on devising a machine-learning DSML. We omit background on MDE and DSMLs, assuming that our readers posses sufficient knowledge in these areas.

2.1 Machine Learning

There is broad agreement that machine learning involves automated pattern extraction from data (Kelleher et al., 2015). The patterns extracted from machine learning techniques are often used by analysts to make predictions. Thus, the most common type of machine learning is supervised machine learning, which is more dominant than unsupervised learning and reinforcement learning (Kelleher et al., 2015). 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_k (Bishop, 2006). For example, consider the case of predicting whether a certain student will gain admittance into university. A natural place to begin is an examination of past admission cycles. We might take in input vectors of student attributes like GPA, test scores, and admission status from the past year. The crucial marker of a supervised learning problem is the inclusion of past observations and their target vectors.

2.1.1 Binary Classification Problems

The most commonly encountered problem classes are binary classification, multiclass classification, regression, and novelty detection (Smola and Vishwanathan, 2008). Our research addresses the binary classification problem, which we describe herein.

Binary classification is perhaps the most understood problem in machine learning (Smola and Vishwanathan, 2008). 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 and rejection are the only classes.

2.2 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 and financially (Costa et al., 2012). Most of the machine learning problem classes are applicable to baseball, and there are a variety of examples of different input variables and predicated variables (Koseler and Stephan, 2018). Some of the predictions are made in real time, for example, guessing the next pitch or where to position the defensive fielders (Baumer and Zimbalist, 2013). Others are made away from the game for example in personnel decisions (Lewis, 2004).

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 (Lewis, 2004). One example of more complex analysis is Bill James's Pythagorean expectation (James, 1987). 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,

$$ExpectedWinRatio = RunsScored^2 / (RunsScored^2 + RunsAllowed^2)$$

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 this paper and the examples we present in the next section are more appropriate for professional analysts and academics interested in the field, as they are more complex and more powerful.

2.2.1 Machine Learning Applied to Baseball

Machine learning's predictive power has led to its use in baseball for both practical and research applications (Koseler and Stephan, 2017a). In general, analyses that employ machine learning improve with increasing numbers of observations (Russell and Norvig, 2003). Due to baseball's relatively large number of observations from 162 games per season with 30 teams, machine learning is a very viable candidate for strong predictive power in baseball.

Our case study focuses on predicting whether a pitcher's next pitch will be a fastball or not. This information is extremely valuable in the context of baseball as it assists a hitter in devising an approach and strategy at the plate. 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 programmed a linear support vector machine classifier to classify pitches based on data from the 2008 season and predict the pitches of the 2009 season (Ganeshapillai and Guttag, 2012). Support vector machines are a type of supervised learning that attempt to find a "separating hyperplane that leads to the maximum margin" (Kelleher et al., 2015), which places classes on different sides of a hyperplane (line), 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.

2.3 Big Data DSML

Based on our research, the only DSML designed to model machine learning was first proposed by Breuker in 2014 (Breuker, 2014). This DSML was designed to bridge the gap between high demand in

industry for Big Data analysts and their lack of software skills. 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 an abstraction of a probabilistic graphical model (PGM). A PGM is a representation of a probabilistic model and is comprised primarily of variables and their relationships. Breuker defines requirements for PGMs to be represented as a DSML to include 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" (Breuker, 2014). Breuker's DSML was built explicitly to work with the Infer.NET [2] 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 omit Breuker's DSML original syntax metamodel from this paper for the sake of brevity and because we provide a representation of it when describing our approach. Breuker provides no accompanying defined code generation scheme, but they do outline a simple skeleton consisting of three methods within a single C# class: *GenerateModel*, *InferPosteriors*, and *MakePredictions* (Breuker, 2014). We thus adapt their concepts as is because they are tailored to work with Infer.NET. By employing their DSML to conduct our case study, we are also validating the DSML by using it in a full end-to-end realization.

3 METHOD

We present an overview of our method in Figure 1. We first form and refine our metamodel in Papyrus. We then create multiple model instances conforming to that metamodel to showcase an ability to update and incrementally create model instances. We then derive and test our code generation engine. Our last step involves us performing code execution and validation experiments. We elaborate on these steps in this section.

[2]https://www.microsoft.com/en-us/research/project/infer.net/

3.1 Metamodel Formation

For our first task, we created a representation of the machine learning DSML in Papyrus (Gérard et al., 2010). 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 2. As Papyrus uses UML on the back-end, the basic elements of our DSML are extensions of the UML element Class. This relationship is demonstrated through the filled triangle arrow. The Node, Gate, GateOption, and Plate elements are the only elements that extend from Class. The Variable and Factor elements are both specializations of Node. Observed Variable and Random Variable are specializations of Variable. This specialization/generalization is represented by the triangle-tipped arrow. Finally, the arrows with an open tip represent different relationships in the Breuker metamodel. These relationships are made explicit through a relationship name which is labeled accordingly. For instance, the relationship between a Plate and a Factor indicates that a Plate selects a factor, and we labeled this "selectsFactor." Barring the Papyrus-required and initial extension 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 eventually create a code generation scheme that is targeted at a binary classification use case only, our Papyrus metamodel will help future users target different use cases and ensure that their model instance conforms to the metamodel. It is available for download on our public university repository. 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. Of course, modelers will have to have knowledge of the machine learn-

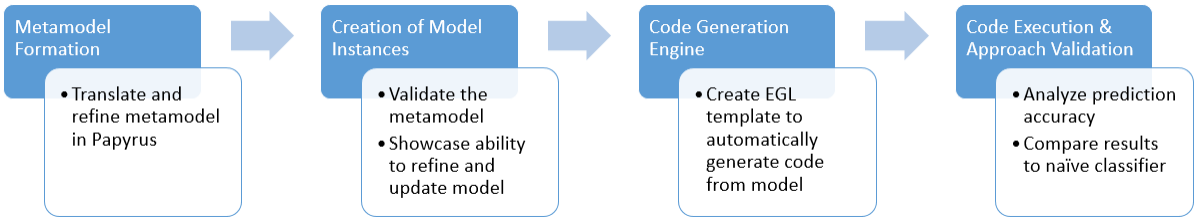

Figure 1: Overview of our Method.

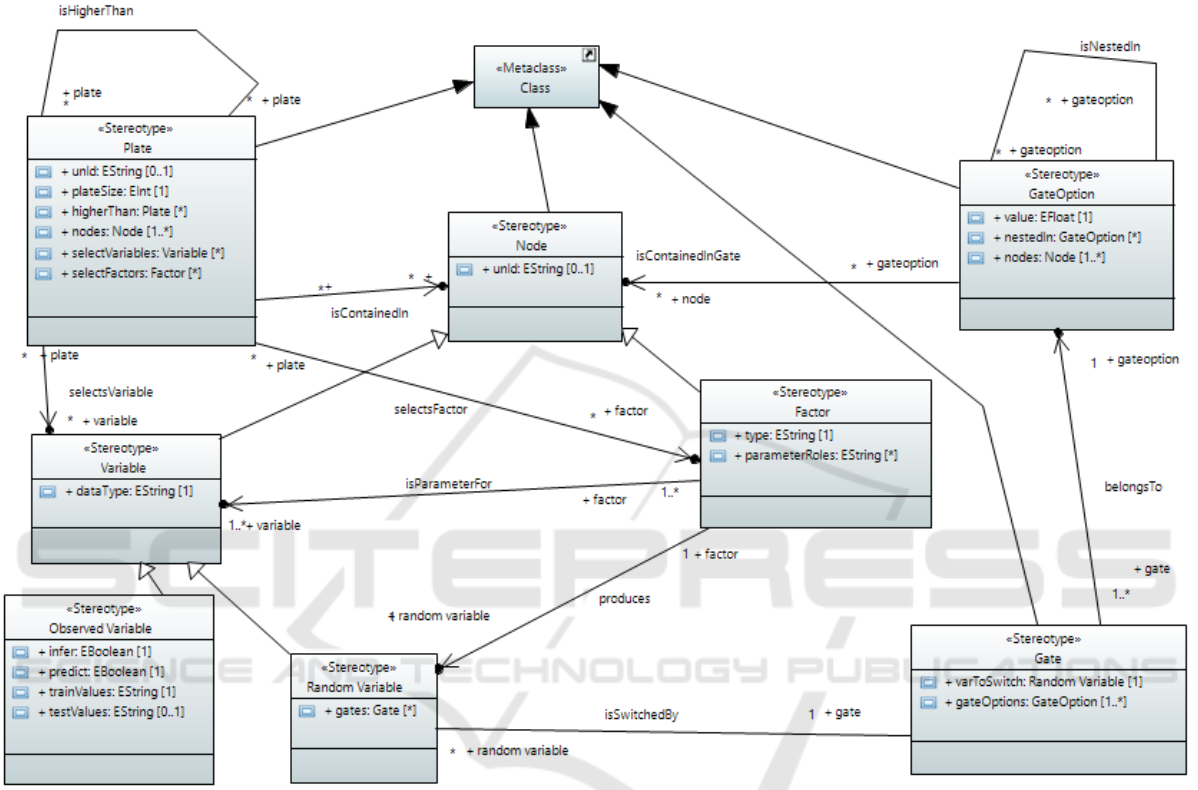

Figure 2: Metamodel in Papyrus.

ing concepts abstracted in the meta model in order to create instance models, however they will not need to develop any code.

3.2 Creation of Different Model Instances

To help validate our DSML metamodel and answer our research question regarding various model instances, we decided to create several iterations 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 will vary in their different features for pitch classification. That is, the purpose of each instance model is to represent the factors considered when deciding whether each observation (pitch) will be a fastball or not. For example, Figure 3 presents a first iteration model instance that uses count (balls and strikes) as the sole predictive factor for the next pitch. A refined version of this model instance we created, which we present in Figure 4, uses the count as before but now includes the pitcher's Earned Run Average (ERA) as a factor in predicting the pitch. For these two examples, we use Papyrus' built-in model validation tool to validate these model instances, finding no errors or warnings, indicating that they conform to our metamodel.

These basic model instances were designed to represent a feasible real-world application of baseball analytics and showcase iterative model development. The use of count and ERA as predictive features is common when evaluating a pitcher's anticipated behavior. While these may be simple examples, they are appropriate for an amateur/novice analyst. These examples also help us confirm that we have correctly implemented our metamodel through

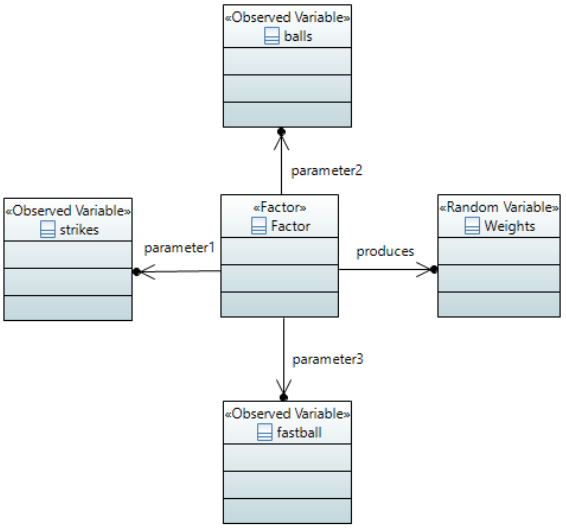

Figure 3: First Iteration Model Instance with Count as Sole Factor (Strikes and Balls).

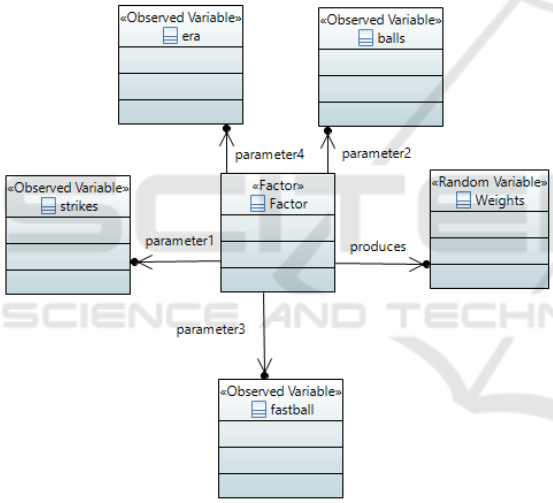

Figure 4: Second Iteration Model Instance with Count and Pitchers ERA as Factors.

Papyrus' built in model-validation functionality. We were able to validate these models manually, and their successful validation through Papyrus indicates metamodel correctness. Through these simple examples, we demonstrate incrementally developing software through small updates to the model instance.

The final iteration of our model instance that we built consisted of 18 factors adapted from the work of Ganeshapillai and Guttag (Ganeshapillai and Guttag, 2012). Due to our inability to obtain the original data source, as we will discuss later, we were limited in our ability to model the original 36 features. The factors we include 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 the glossary of our thesis (Koseler, 2018). We present our full model instance in Figure 5. In the interest of readability, we do not display the names of the relationships between the constructs in the model instance and we condense the model elements. The relatively large number of predictive features makes it difficult to create an aesthetically pleasing and readable model for us to present in this paper. However, the model navigator in Papyrus allows for easier interpretation, and, in either case, models are better suited for readability than source code. Users can download these models on our public repository to view in Papyrus. In the next section, we discuss the code generation engine we designed that automatically generates code from these model instances.

3.3 Devising the Code Generation Engine

In this step, we devised 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 from scratch as Breuker provided no code nor examples, 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 Epsilon Coordination Language (EGX), EGL (Epsilon Generation Language), and the Epsilon Object Language (EOL) (Kolovos et al., 2006). Our code generation template is comprised of eighty four lines of code. This will automatically generate source code totaling roughly eighty four lines of code that makes calls to the Infer.NET library.

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 the plausibility of using MDE to create machine learning software without writing code manually.

We have uploaded our code generation template to our public university repository along with the other artifacts we generated for this project. In the Epsilon family of languages, text in the code template that is enclosed in "[%" brackets is dynamic and variable based on the user-defined model instances. All other

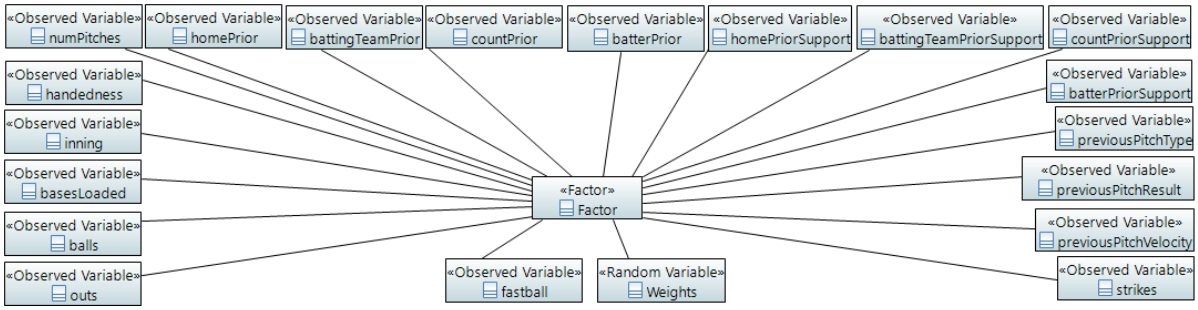

Figure 5: Complete Model Instance with 18 Factors.

text in the template is static. Our first step in our code generator 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 (Minka, 2001) 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 (pitch), 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.

3.3.1 Quality of Generated Code

The quality of the generated code is important because the code generated in MDE techniques must be correct and robust since users will interact with modeling artifacts only, not code. To help assess the quality of the generated code, we used input space partitioning (Ammann and Offutt, 2016) to test the code by devising several different inputs as training and test data. For instance, the test case of using valid integers for the "strikes" data that do not make sense in the context of baseball, such as a value of 10 where there can be only a value of 0,1, or 2 strikes. Overall, we found the quality of the code to be fairly high, with some important areas for improvement. In particular, a mismatched number of training observations for each Observed Variable is allowed by the program. If a user inputs 3 values for one Observed Variable and 4 for another, the code will still execute when it should throw an exception. 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. Given that Papyrus allows for such limits on entered values, our MDE method need not be limited to the realm of baseball analytics. These limits can be adjusted as appropriate depending upon the domain.

3.4 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, and thus did not pitch during the 2017 season. We retrieved this data from Baseball Savant, which provides official Major League Baseball data available publicly for free [3]. 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 to allow for reproduction of our experiments. 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 data en-

[3] https://baseballsavant.mlb.com

try 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 $\{x_1, x_2, \ldots, x_n\}$, where $n = NumberOfObservations$. Thus, we used Python and the Pandas library[4] to read our data, which is in the form of a comma separated values (CSV) file.

To help validate our model and the process as a whole we fed this formatted data to our complete model instance in Papyrus, subsequently generating an executable C# file. After running this 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. We then compared it against our actual test data from 2017. We further compared our predictions against 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.

4 RESULTS AND DISCUSSION

In this section, we present the quantitative results of using our DSML full-factored model instance and automatic code generation MDE solution to this baseball analytics binary classification problem. This mimics the process that analysts can follow now that we have created meta models and code generation engine. We followed the steps that an analyst would take, that is, formatting our data, building the instance model conforming to the meta model, feeding the data into the instance model, and running our automatically generated software. In doing so, our software exhibited a prediction accuracy of 71.36%. That is, 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. In contrast to a traditional programming based solution, this DSML system helped the modeler in that Infer.NET calls were invoked through an easily generated and modifiable instance model conforming to a meta model, not requiring a user to deal with source code at all.

While our prediction accuracy is higher than Ganeshapillai and Guttag, we must note that our model exhibited a smaller increase over the naive classifier than their work. 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 over the naive classifier. 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 executed and gave us an improvement in prediction accuracy over a naive classifier. Figure 6 illustrates the prediction accuracy of our work versus that of Ganeshapillai and Guttag.

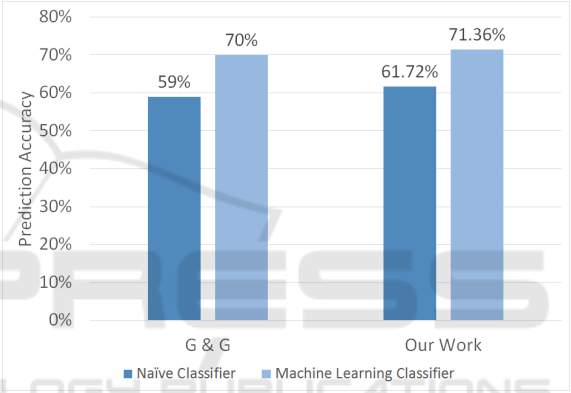

Figure 6: Prediction Accuracy for our Work and Ganeshapillai and Guttag's.

4.1 Research Questions

At the beginning of this project, we set out to answer two main research questions. Regarding our first question, we were able to successfully encapsulate machine learning binary classification concepts in a DSML and devise an accompanying code generation scheme. Although we demonstrated this with a baseball analytics use case, our DSML metamodel is general in that it 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 can support a number of features and observations. For the second part of our first research question, we encountered several challenges which we document in Section 4.3.

For our second research question, we realized a complete application of MDE for machine learning that allows for model updates/refinements and code generation through a baseball analytics use case.

[4]https://pandas.pydata.org/

We created some iterations of model instances to demonstrate 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. In our solution, domain experts need to have knowledge of machine learning concepts only, rather than machine learning concepts and source code development skills. We have essentially raised the level of abstraction for analysts to develop solutions.

4.2 Threats to Validity

Most prominently, our training and test data is different than the set that Ganeshapillai and Guttag (Ganeshapillai and Guttag, 2012) used for their research. 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 data owners, and they were unwilling to provide us access to that data. The data was available for purchase but at a prohibitively expensive cost as the data owners had changed their business model since Ganeshapillai and Guttag's original experiments. Since we were concerned mainly with the plausibility of using MDE for this binary classification problem and exhibiting only comparable results to the traditional coded approach, we decided having their exact data was unnecessary and that real-life baseball data, albeit from a different year, was acceptable. However, it is still a threat to validity. It may be the case that had we used their older data set, 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 and the game in general.

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. For example, we did not have access to play-by-play statistics for each batter that was facing 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 the necessary onerous web querying required to gather data on the free Baseball Savant website, we decided to use 4 MLB teams as our baseline rather than the entire league, resulting in 85,000 observations. We used the New York Mets, New York Yankees, Toronto Blue Jays, and Cincinnati Reds as these were teams of personal 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 a significant result in itself.

Another important threat to validity is our use of a different algorithm for making predictions. Ganeshapillai and Guttag used a Support Vector Machine to classify their pitches. Because we were extending a DSML that is 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 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. This represents a limitation of our DSML.

Finally, our metamodel and code generation engine were built to work with Papyrus. We chose 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 included a ready-to-import Papyrus project in our public repository that can be used directly in Papyrus or converted for other tools.

4.3 Lessons Learned and Challenges

Part of our goal in our industrial case study was to help identify lessons that we learned and interesting challenges.

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 (Ganeshapillai and Guttag, 2012). 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. 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 lesson, 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. Another obstacle we faced in using Papyrus was the method required to feed data to the model. As we discussed in Section 3.4, there is no systematic way to feed large strings of information to the model, so it must be done manually.

4.4 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 nor 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 was able to support a fair amount of features and training observations.

An important impact of this work is that we published our code generation engine, models, and other artifacts on our public repository, and they 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 practical impact of this work. While one can have doubts about the representatives of baseball analytics for industrial practitioners, future practitioners or academics 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 optionally address the Gate, GateOption, and Plate constructs that were unnecessary for our case study. Although we chose to demonstrate the feasibility of MDE in the machine learning domain through a baseball analytics case study, future researchers can apply these same techniques to different problems and data sources. We plan on doing so to measure the generality of this work to other applications and problems, and to facilitate further understanding and use. The code generation we developed is problem-agnostic; as long as the data is formatted properly for a binary classification problem, the generated software can derive meaningful predictions.

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. Additionally, we could quantify how much effort a domain expert would require to address a relevant domain data problem, and how much they enjoyed this MDE-based method. Lastly, we could also compare it to some of the approaches we describe in our Related Work section using the same dataset to better explicate its advantages and benefits over other methods. We deemed this beyond the scope of the project at this time.

5 RELATED WORK

To our knowledge, while there is some work on machine learning domain specific languages (DSL) (Portugal et al., 2016), there is little other related work in creating a machine learning DS**M**L (Zafar et al., 2017). The paper by Breuker (Breuker, 2014) 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 that allow for rapid application of 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 (Demšar et al., 2013). The crucial difference between our work and Orange is that Orange does not adhere to nor support the MDE approach to formal 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 data, assess prediction accuracy, and build visualizations. There are other similar UI-based editors for machine learning work flows, but they do not follow a strict model-driven approach.

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 (Hall et al., 2009). 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 (Abadi et al., 2016). 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 an MDE 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.

6 CONCLUSION

In this paper, we considered the plausibility of using Model-Driven Engineering to build machine learning software. Much like Breuker (Breuker, 2014), we were motivated by the industrial problem of having a high demand for machine learning expertise and a shortage of individuals with the technical knowledge required to build such quality 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 and refined 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 model instances in Papyrus and have the native model validation tool ensure that their model instances are in accordance with the metamodel. To confirm this, we built several iterations of model instances of the baseball analytics problem of classifying pitches as fastball or non-fastball based on past data. Our early iteration model instances allowed us to determine the ease of quickly and incrementally building and defining our software through models. Further, we built a code generation engine using EGX, EGL, and the Epsilon Object Language that allows users to generate code based on their model instances. Our case study 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 Guttag (Ganeshapillai and Guttag, 2012), we successfully achieved a comparable increase in prediction accuracy over a naive classifier. Most importantly, we did this through automatically generated code that we used to make predictions. Our application of this MDE solution to an industrial machine learning context provides evidence of a complete application of machine learning MDE, and helps prove the viability of MDE in the machine learning domain. We believe the practical impact of this work includes helping facilitate future MDE machine learning practice and research through extension of, and drawing inspiration from, our described process and the artifacts we published on our repository.

ACKNOWLEDGEMENTS

This work is supported by the Miami University Senate Committee on Faculty Research (CFR) Faculty Research Grants Program.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zheng, X. (2016). Tensorflow: A system for large-scale machine learning. In *Operating Systems Design and Implementation*, OSDI'16, pages 265–283, Berkeley, CA, USA.

Ammann, P. and Offutt, J. (2016). *Introduction to software testing*. Cambridge University Press.

Baumer, B. and Zimbalist, A. (2013). *The sabermetric revolution: Assessing the growth of analytics in baseball*. University of Pennsylvania Press.

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

Breuker, D. (2014). Towards model-driven engineering for big data analytics–an exploratory analysis of domain-specific languages for machine learning. In *Hawaii International Conference on System Sciences*, pages 758–767. IEEE.

Costa, G. B., Huber, M. R., and Saccoman, J. T. (2012). *Reasoning with Sabermetrics: Applying Statistical Science to Baseball's Tough Questions*. McFarland.

DeLine, R. (2015). Research opportunities for the big data era of software engineering. In *International Workshop on Big Data Software Engineering*, pages 26–29.

Demšar, J., Curk, T., Erjavec, A., Črt Gorup, Hočevar, T., Milutinovič, M., Možina, M., Polajnar, M., Toplak, M., Starič, A., Štajdohar, M., Umek, L., Žagar, L., Žbontar, J., Žitnik, M., and Zupan, B. (2013). Orange: Data mining toolbox in python. *Journal of Machine Learning Research*, 14:2349–2353.

Ganeshapillai, G. and Guttag, J. (2012). Predicting the next pitch. In *Sloan Sports Analytics Conference*.

Gérard, S., Dumoulin, C., Tessier, P., and Selic, B. (2010). 19 Papyrus: A UML2 tool for domain-specific language modeling. In *Model-Based Engineering of Embedded Real-Time Systems*, pages 361–368. Springer.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., and Witten, I. H. (2009). The WEKA data mining software: an update. *SIGKDD Explorations*, 11(1):10–18.

James, B. (1987). *The Bill James Baseball Abstract 1987*. Ballantine Books.

Kelleher, J. D., Mac Namee, B., and D'Arcy, A. (2015). *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies*. MIT Press.

Kelly, S. and Tolvanen, J.-P. (2008). *Domain-specific modeling: enabling full code generation*. John Wiley & Sons.

Kent, S. (2002). Model driven engineering. In *International Conference on Integrated Formal Methods*, pages 286–298. Springer.

Kolovos, D. S., Paige, R. F., and Polack, F. A. (2006). The epsilon object language (eol). In *European Conference on Model Driven Architecture-Foundations and Applications*, pages 128–142. Springer.

Koseler, K. (2018). Realization of Model-Driven Engineering for Big Data: A Baseball Analytics Use Case. Master's thesis, Miami University, Oxford, Ohio, USA.

Koseler, K. and Stephan, M. (2017a). Machine learning applications in baseball: A systematic literature review. *Applied Artificial Intelligence*, 31(9-10):745–763.

Koseler, K. and Stephan, M. (2017b). Towards the realization of a DSML for machine learning: A baseball analytics use case. In *International Summer School on Domain-Specific Modeling Theory and Practice*. https://sc.lib.miamioh.edu/handle/2374.MIA/6224.

Koseler, K. and Stephan, M. (2018). A survey of baseball machine learning: A technical report. Technical Report MU-CEC-CSE-2018-001, Department of Computer Science and Software Engineering, Miami University. https://sc.lib.miamioh.edu/handle/2374.MIA/6218.

Lewis, M. (2004). *Moneyball: The art of winning an unfair game*. WW Norton & Company.

Mattson, M. P. (2014). Superior pattern processing is the essence of the evolved human brain. *Frontiers in neuroscience*, 8.

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Uncertainty in Artificial Intelligence*, pages 362–369, San Francisco, USA.

Portugal, I., Alencar, P., and Cowan, D. (2016). A preliminary survey on domain-specific languages for machine learning in big data. In *SWSTE*, pages 108–110.

Russell, S. J. and Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition.

Sawchik, T. (2015). *Big Data Baseball: Math, Miracles, and the End of a 20-Year Losing Streak*. Macmillan.

Smola, A. and Vishwanathan, S. (2008). *Introduction to Machine Learning*. Cambridge University Press.

Witten, I. H., Frank, E., Hall, M. A., and Pal, C. J. (2016). *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann.

Zafar, M. N., Azam, F., Rehman, S., and Anwar, M. W. (2017). A systematic review of big data analytics using model driven engineering. In *International Conference on Cloud and Big Data Computing*, pages 1–5.

Zimmerman, A. (2012). Can retailers halt 'showrooming'. *The Wall Street Journal*, 259:B1–B8.