ML2ESC: A Source Code Generator to Embed Machine Learning Models in Production Environments

Oscar Castro-Lopez, Ines F. Vega-Lopez
Universidad Autonoma de Sinaloa, Culiacan, Sinaloa, Mexico

Abstract—Deployment of Machine Learning (ML) models into production systems is a complex and error prone task. A trained ML model is a formal specification, with a compiler we can transform it to source code. We propose a compiler to translate ML models to source code, automating the deployment task. The resulting code can be embedded inside operational environments to provide real-time predictions to the end-user with an efficient execution of the deployed ML models. We have already implemented this idea in a prototype that translates ML models from R models and the PMML standard to various target languages.

Keywords: Machine Learning Engineering, ML Deployment, PMML

1. Introduction

Nowadays, the interest of using machine learning (ML) in software systems has increased considerably. The life cycle of software development that integrates ML models starts with a team of data scientists using modeling tools to build a mathematical function that describes the relationship between an attribute (dependent variable) of an object in a sample with respect to one or more attributes (independent variables) of the same object. These tools provide a domain-specific language to manage and process data and are designed for a rapid building and prototyping of ML models.

After testing and validation, the ML models must be deployed in the operational environment where value can be drawn with the predictions or inferences that can be obtained on new input data. ML models deployment is the general process of taking a model (mathematical function) to a specific operating environment where it is available for use (Software). Therefore ML models are handed out to software engineers to integrate them as part of the operational software, this task is treated as any other software engineering task. We argue that deploying ML models is not a conventional software engineering task.

For one, the technical language between ML modeling and software engineering is very different and the complexity of ML models is very demanding, coding the logic and math behind predictive models is a hard task even for the most experienced software engineers [1]. The modeling environment and the software development environment are very different, and the deployment process and know-how sometimes is fragmented into documents, code or even only in people’s heads, where knowledge can easily be lost [2].

Perhaps the most important characteristic, is that ML models can be formally defined, after all they are mathematical formulae. The ML model constructed in the modeling process is a formal specification of a model that is going to be integrated somehow with our software. From a formal specification of a ML model, it is possible to transform it algorithmically (ie compile) to a high-level language. This task can be done with a compiler, given a sequence $S$ which represents a formal specification of our ML model and $T$ is a target language, we want a transform it to $S'$ which is a sequence of the source code in our target language. Compilers perform repetitive and formally defined tasks, that otherwise done manually would require a lot of effort and be prone to errors.

We propose to automate the coding task with a compiler for the easy deployment and integration of predictive models with operational systems. Using as an input a standard description of a predictive model, and having as an output the source code in a target programming language. The compiler has a multi-target approach in order to be able to generate source code in different programming languages accordingly with the operational environment. The resultant source code is required to be semantically equivalent to the original.

The main contributions of this paper are the following.

- We propose the use of a compiler to translate a standard description of a ML model to source code in a target language.
- The use of this compiler can significantly reduce the time-to-deploy of ML models by automating a labor-intensive task and eliminating human induced errors.
- We have released a proof of concept of this compiler for Generalized Linear Models (GLM) however our compiler currently supports Neural Networks (NN) and Support Vector Machines (SVM) as well.

The rest of this paper is organized as follows Section 2 is a review of the related work in ML models deployment, Section 3 presents the general design of the compiler and our proof-of-concept, finally Section 4 presents the conclusions of this paper.

2. Related Work

ML model deployment is the connection between predictive modeling and operational environments, it is constrained by the specifications and requirements of the operational software. Grossman describes in [3] some common approaches to deploy ML models, this list is very similar to ours, there are at least the following deployment approaches: modeling tools pipelining, model consumer servers, enterprise solutions servers, and in-house coding.
Modeling tools pipelining in this case both modeling and deployment is done in the same tool, where we need to communicate operational software with modeling tools. This requires a running server available with the modeling tool, packages, dependencies, and ML models loaded. For example the most popular modeling tools used as the back-end to pipeline ML models are: Scikit-learn/Python [4], R [5], Knime [6], Weka [7]. Since the deployment task is a business driven decision, there is not much documentation available for a general deployment process. We can find many scientific papers where ML models were implemented using one of those tools. There are many R packages that allow connection to other applications that allow using R as a back-end. For example Malviya et al. in [8] proposes R as an analytic framework and deployment platform. Another example is the use of Flask which is described as a microframework for python to enable web development and can be used to deploy ML models built in python. For example Kühl et al. [9] describes a complete workflow of modeling and deployment with python.

Model consumer servers are specialized tools for deployment, where ML models described in standard formats are imported in a server, Predictive Model Markup Language (PMML) [10] and Portable Format for Analytics (PFA) [11] are the most popular standards. These tools load and interpret the model stated in the standard format, the operational software sends the data to the server to make predictions and the server returns the predictions. There are many tools compatible with the PMML standard, many commercial and open-source tools listed in the Data Mining Group (DMG) official website follow a similar approach and consume PMML files to deploy ML models in a server. For example recent works such as Ferguson et al. in [12] proposes a scoring engine in the cloud and a microcontroller such as Raspberry PI for smart manufacture, or Heit et al. in [13] proposes the Model Deployment and Execution Framework (MDEF), this tool leverages PMML to deploy models in Java and Scala applications. In the case of PFA Pivarski et al. in [11] proposes Hadrian, a scoring engine that compiles PFA PMML files to deploy ML models in a server. For example we know Ferguson et al. in [12] proposes a scoring engine in the cloud and a microcontroller such as Raspberry PI for smart manufacture, or Heit et al. in [13] proposes the Model Deployment and Execution Framework (MDEF), this tool leverages PMML to deploy models in Java and Scala applications. In the case of PFA Pivarski et al. in [11] proposes Hadrian, a scoring engine that compiles PFA PMML files to deploy ML models in a server. For example Ferguson et al. in [12] proposes a scoring engine in the cloud and a microcontroller such as Raspberry PI for smart manufacture, or Heit et al. in [13] proposes the Model Deployment and Execution Framework (MDEF), this tool leverages PMML to deploy models in Java and Scala applications. In the case of PFA Pivarski et al. in [11] proposes Hadrian, a scoring engine that compiles PFA documents into Java bytecode on-the-fly for its deployment.

Enterprise solutions are also a popular option, this refers to a full framework to build and deploy predictive models in a server. There is a considerable list of companies that allow to model and deploy predictive models with their services in the cloud. Most of the time it is necessary to obtain a proprietary software license along with hardware, or to rent cloud infrastructure. The actual deployment task may vary but it can be via RESTful API, Web Services or direct communication through different protocols. Many commercial software solutions are centered in this approach, for example: SAS, Stata, SPSS, AzureML, etc.

In-house coding requires that a team of software engineers collaborate with data scientists to fully understand and manually code predictive models in the programming language of an operational environment. It also requires that the data scientists test and validate that the output given by the translated model is the same as the original.

The deployment options where a server is involved requires building a pipeline between machine learning tools and operational software. This “client-server” like approach has a one single point of failure and in some operational environments may not be feasible due to networking restrictions or simply because it is required to be a standalone application. A pipeline can be a complex piece of the system and become difficult to maintain and test over time. Modeling tools in general are not designed to handle many requests or to be scalable. In this case the ML models are loaded in memory, the model is interpreted by the server, which in applications where memory management and low latency are key requirements it is not the best option.

We think that for software applications where efficiency and speed matter, implementing these predictive models in the same programming language than the operational system may be a better approach. However, manually translating a ML model into source code is a cumbersome and error-prone task. We argue that this is a task that can be easily automated by the use of computer science tools such as the proposed compiler.

3. ML2ESC Compiler

Our approach mainly proposes using as an input a standard specification of a ML model, in doing so, we are not limiting the data scientist to the use of a particular tool. A general process to compile conventional machine learning models. The output of our compiler is pure source code to directly embed it in the operational software. We believe that by having the source code available, software engineers can perform a fast and efficient deployment process and prediction execution. The design of our compiler is multi-target, having many target languages to be able to integrate it easily with the software running in the operational environments. Of course we cannot commit to target all possible languages, but our design is robust enough to incrementally add new target languages.

We do not intend to build a general purpose compiler, we only require a subset of the languages to be able to express the ML computations, therefore we avoid several intermediate layers and tasks present in classical compiler workflows. Thus, we avoid a lot of the general complexity of a full general purpose compiler. Nevertheless our approach embraces compiler techniques such as parsing, building intermediate representation and the use of templates to generate source code. We have a more focused and defined spectrum to customize and generate efficient code for the different target languages. Overall, we already know the general structure of the prediction algorithm where we need to fill with the specific data of the already trained ML model. Therefore we can achieve high efficiency for the set of operations required for our generated source code.

We propose the Machine Learning to Embed Source Code (ML2ESC) compiler, the input of this tool is a standard specification of a ML model. Our front-end is not a typical
parsing task reading code, we read a formal specification, and we need to extract the relevant information of the ML model. For example for neural networks the layers, neurons, connections, weights, activation functions, the network structure, etc. The extracted information is the input to build our intermediate representation (IR). From our IR we translate to the target language. We can see the general structure of the compiler design in Figure 1.

**Front-End:** We propose to use standard formats of ML models for the front-end of the compiler. This allows data scientists to carry out their modeling process in the tool they choose and export a trained ML model to a standard format such as PMML or PFA. Our front-end reads and extracts the useful information from the ML specification to translate the input into our intermediate representation. The compiler design is flexible enough to allow adding modules to the front-end to translate from other formats. Note that since this process is machine to machine, the syntax verification is simplified.

**Intermediate Representation:** As stated before we do not need to have all the abstractions of a full programming language, we need a sufficiently expressive IR to support the algorithms for ML models, and an IR that maintains enough information to be able to translate it to the target language.

Our intermediate representation is essentially a tree structure, with a set of language elements to construct the code. We can define an intermediate representation template for each ML model, this is because the base structure of the algorithm does not change, just some specific details change. For example in the case of a Generalized Linear Model (GLM) we know that we will define two functions, in the first function basically a dot product of two vectors is computed, and in the second a link function such as logit is applied to the result of the first function. After parsing a ML model we can instantiate our IR templates for the specific model to produce the IR.

Each node used to define the Abstract Syntax Tree (AST) graph is a language element part of our IR. For example in Figure 2 we illustrate an IR with an AST of a GLM prediction algorithm, the base structure of the AST is the same for all GLM models, what varies are some elements of the structure or depth of the tree.

Linear algebra operations are used in almost every ML model computations, therefore we defined them as math expressions in our IR, as it can be seen in the rightmost node in Figure 2, the dot product is the main operation of a GLM. If we expand a dot product in an AST it would look something like Figure 3. The dot product is also used in Neural Networks, Support Vector Machines, and many other models; therefore is present in our defined AST templates, we only show the GLM template due to size limitations of this paper.

**Back-End:** In the back-end we have several modules, one for each target language. The input of the back-end is the IR. In each module we navigate through the tree to generate source code from our IR that will become our output, source code in the target language of the specific module.
Table 1: Linear Algebra Expressions for vectors x and y that are part of our Intermediate Representation.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDITION</td>
<td>$x + y = (x_1 + y_1, x_2 + y_2, ..., x_n + y_n)$</td>
</tr>
<tr>
<td>SUBTRACTION</td>
<td>$x - y = (x_1 - y_1, x_2 - y_2, ..., x_n - y_n)$</td>
</tr>
<tr>
<td>SCALING</td>
<td>$\alpha x = (\alpha x_1, \alpha x_2, ..., \alpha x_n)$</td>
</tr>
<tr>
<td>NORM</td>
<td>$</td>
</tr>
<tr>
<td>DOT</td>
<td>$x \cdot y = x_1 y_1 + x_2 y_2 + ... + x_n y_n$</td>
</tr>
</tbody>
</table>

resulting code is semantically equivalent to the specifications of the input ML model. With the flexible design of our compiler we can easily add modules for a specific target language, such module translates our IR to source code in order to embed the ML model in the operational software. The generated source code is ready to be compiled or used, it does not require editing by hand.

Proof-of-concept: We already have a proof-of-concept implementation of our compiler approach. We built the glm.deploy package [14], which is already available on CRAN. This package takes as an input a trained GLM object and compiles the source code to either C or JAVA. The resulting C code is ready to be called and compiled to be used in the target operational machine. The resulting JAVA code is ready to be called from a class as a static function, and it can be used in many java compatible applications. In this way with the statistical software R we can build a model using the GLM function. Once the model is built we can export it to our target language and integrate it with the operational environment to deploy the model in production.

The glm.deploy package was our initial step to prove the feasibility of our approach. We are currently working on a more ambitious compiler where we can use as an input a standard format for ML models. In our work in progress the front-end translates a ML model defined in PMML to the Intermediate Representation, and the back-end translates the IR to source code to integrate into operational software. The ML models can be built in many modeling tools as long as they are compatible to export them to PMML. Currently we support the following models: Generalized Linear Model, Neural Networks, and Support Vector Machines. As target languages we currently support: C, Java, SQL, and we are starting to work with CUDA.

With this approach we can build the ML models in many different tools and export them to the PMML format. The PMML file is used as the input to our compiler which parses the file and extracts the information needed to build the IR which already has a tree template for each model type, the IR is passed to our back-end which generates the source code in the target language.

4. Conclusion

The ML2ESC compiler generates source code from ML learning models described in a standard format. The generated source code is ready to be embedded in operational environments, and allows to automate the deployment process in ML projects. The compiler is a useful resource to bridge the gap between ML modeling and a final software product infused with ML. ML2ESC is particularly important to help building the ML part of applications. This task is expensive to perform because the math and knowledge that is behind the ML algorithms are complex and its construction and maintenance are difficult. Software products with ML capabilities must be free of errors, efficient and portable with minimal dependencies. To be able to have the ML model in source code, for example in C/C++, allows a more precise memory management and compiled code can run several times faster than interpreted code.

Acknowledgment

The authors would like to acknowledge the funding provided for this research by the Mexican Council of Science and Technology (CONACYT).

References