2021 ACM/IEEE International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C) | 978-1-6654-2484-4/21/$31.00 ©2021 IEEE | DOI: 10.1109/MODELS-C53483.2021.00028

A Model-Driven Engineering Approach for Monitoring Machine Learning Models

Panagiotis Kourouklidis
British Telecom
University of York
Ipswich, United Kingdom
panagiotis.kourouklidis@bt.com

Dimitris Kolovos
University of York
York, United Kingdom
dimitris.kolovos@york.ac.uk

Joost Noppen
British Telecom
Ipswich, United Kingdom
johannes.noppen@bt.com

Nicholas Matragkas
University of York
York, United Kingdom
nicholas.matragkas@york.ac.uk

***Abstract*—Once a machine learning (ML) model is produced and used for commercial purposes, it is desirable to continuously monitor it for any potential performance degradation. Domain experts in the area of ML, commonly lack the required expertise in the area of software engineering, needed to implement a robust and scalable monitoring solution. This paper presents an approach based on model-driven engineering (MDE) principles, for detecting and responding to events that can affect a ML model's performance. The proposed solution allows ML experts to schedule the execution of drift detecting algorithms on a computing cluster and receive email notifications of the outcome without requiring extensive software engineering knowledge.**

***Index Terms*—Machine Learning, Model-Driven Engineering, Dataset Shift, Concept Drift, Data Drift**

I. Introduction

As usage of machine learning (ML) techniques for commercial purposes becomes increasingly commonplace, a lot of research has focused on developing algorithms for detecting changes in the environment in which a ML model operates and adapting to them. Applying such algorithms is vital to ensure that the performance of a deployed ML model stays adequate in a changing environment. Unfortunately, just developing a suitable algorithm is not enough in a commercial setting as there is also considerable engineering effort required to architect a system that can meet the performance requirements of the business.

This paper presents a model-driven approach for ML monitoring. The extent to which such an approach can lower the technical barriers that ML experts face when designing ML monitoring systems is explored. A proposed solution is shown that allows ML experts to define the intended behaviour of certain ML monitoring systems without having to provide their technical implementation.

The rest of the paper in structured as follows: First, a brief overview of the ML monitoring domain is provided. Afterwards, an initial meta-model of the domain is presented. Then, the generation of a technical implementation according to a model-driven engineering (MDE) model of a ML monitoring system is shown. Next, a web application that enables domain experts to experiment with the proposed solution is shown. Finally, the paper concludes with a discussion about planned future work.

II. Domain Analysis

Techniques that can infer the value of a target variable Y, when given the value of an observed variable X, have practical applicability in numerous scenarios, such as spam detection, fraud detection and digital marketing. Developing such techniques, has therefore been the focus of various scientific fields, such as statistics, pattern recognition and more recently, ML. This paper focuses on a subset of ML techniques called supervised learning. The scenario in which supervised learning is applicable is the following. One would like to predict the value of a variable Y when given the value for a variable X but there is no known mapping between them. Supervised learning algorithms seek to extract a mapping between X and Y from a set of labelled samples, known as the training set. Once the mapping is extracted, it can be used to infer the value of Y for unlabelled samples, known as the test set.

In the supervised learning literature [1], [2], X and Y are often treated as random variables that follow a joint probability distribution P(X,Y). Regardless of the algorithm used to obtain the mapping from X to Y, samples in the test set need to be drawn from the same joint probability distribution as the ones in the training set in order to ensure good predictive performance. This assumption might not be true for a number of reasons. The real world is a complex, non-stationery environment and thus differences in the joint probability distributions between training and test sets should be expected, especially over time. This phenomenon, in its numerous variations, is well-studied in the literature under names such as concept drift/shift [3], [4], covariate/sampling shift [4]–[6], prior probability shift [4], [6] and the more general, dataset shift [6], [7].

Methods that adapt to such non-stationery environments are also well-studied. As an example, in [6] the author presents a methodology for adapting a trained model using the marginal probability distribution of X in the test set. In [4], the authors propose a number of techniques for counter-acting drift depending on whether some of the labels for samples in the test set are available or not, and under the assumption that we know the causal model of the phenomenon we try to model.

An alternative approach is followed by a subset of machine

978-1-6654-2484-4/21/$31.00 ©2021 IEEE
DOI 10.1109/MODELS-C53483.2021.00028

learning called online learning [2]. These algorithms do not make any assumptions about the underlying distribution of observed and target variables and continuously adapt for every labelled sample received. Therefore, online learning methods can be used when one expects that the environment will change over time. The main disadvantage of this approach is that it needs a constant flow of labelled samples which is not always possible.

A hybrid approach is also possible. In this approach, the model is trained using an initial labelled dataset but also adapts whenever a new batch of labelled samples becomes available. In [8], the authors suggest the use of an ensemble of models to achieve good performance even in the presence of drift. Whenever a more recent labelled dataset becomes available, an additional model is trained and added to the ensemble. Then, the algorithm checks whether the ensemble's performance has stayed consistent. Depending on the outcome there is an adjustment to the weights of the models in the ensemble.

III. Domain Meta-Model

As presented in the domain analysis, research scientists have developed a number of ML techniques that achieve good predictive performance even when deployed in a non-stationary environment. However, in order for these techniques to be applied in a commercial setting, they would have to be part of a system that meets the requirements of various stakeholders. For example, such systems might need to consider latency, cost of computational and storage resources as well as compatibility with pre-existing software and infrastructure. The issue is further complicated by the fact that the person that designs the methodology for learning in a non-stationary environment is not necessarily the person responsible for the technical implementation of the overall system. This introduces an additional layer of challenges that can happen due to miscommunication between people.

This work investigates whether the application of MDE principles can reduce the overall effort required to implement and deploy ML based systems that can handle non-stationary environments. A critically important part of the solution is the design of a domain-specific language (DSL), that can express as many ML monitoring methodologies as possible while being minimally bound to the technical implementation of the underlying system. The DSL aspires to provide a standardised communication layer between domain experts who declaratively define the behaviour of the ML monitoring system and software engineers who are responsible for generating a concrete implementation that adheres to the specified behaviour.

This section presents an initial effort to design such a DSL. Specifically, the meta-model of a ML-based system that is designed to handle various types of non-stationarity is presented. In addition, the assumptions behind the various design decisions and some planned future improvements are discussed.

The assumptions about the environment in which a deployed ML model operates are as follows:

- A deployed ML model continuously receives unlabelled samples (inference requests) and is expected to return the predicted labels (inference responses) as soon as possible.
- After an unspecified amount of time, the actual labels will be made available for a portion of the unlabelled samples previously received.
- Periodically, a decision shall be made on whether to take corrective action based on the latest data received by the ML model. Both labelled and unlabelled data could be used to make this decision depending on the chosen methodology.
- There are a number of different actions that can be taken. It is up to the domain expert to define which action should be taken as a response to different kinds of drift.

Taking the above assumptions into account, it is proposed that the meta-model that can describe ML monitoring methods should have the following essential components:

- A mechanism to specify which ML model one wants to monitor.
- A mechanism to specify which fields of the inference request and response are to be captured. Also a way to name the captured fields so that other entities can reference them individually.
- A mechanism to define periodic drift detection executions.
- A mechanism to define what actions are to be taken when drift is detected.

Following the above general ideas, a preliminary version of a meta-model for ML monitoring is presented. Although the goal is to design a meta-model that is completely agnostic to any underlying technical implementation, some compromises have been made in this version in the interest of getting feedback from domain experts sooner. These compromises will be pointed out and a discussion about how they can be rectified will be presented in the Future work section.

Figure 1 shows the classes of the meta-model and how they relate to each other. Here is a description of what each class represents:

A. Deployment

A *Deployment* is the top-level class of the meta-model that describes all the aspects of a ML monitoring method. It contains exactly one instance of the class *Model* and any number of instances of class *DriftDetector*.

B. Model

A *Model* represents the ML model that the domain expert wants to monitor and it contains general information about it. Specifically, it contains the name of the ML model, the URL where a serialization of the ML model can be found and the name of the framework that was used to create it (e.g TensorFlow[1]). In the interest of simplifying demonstrations

[1]https://www.tensorflow.org

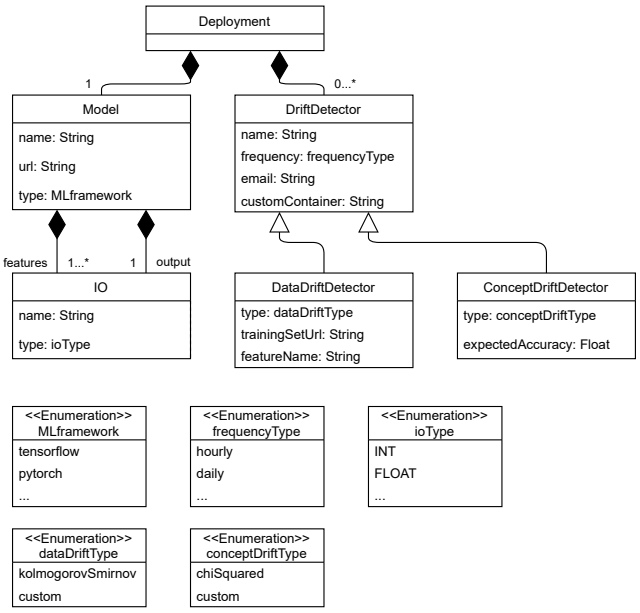

Fig. 1. Domain Meta-Model

of this work, the specified ML model is deployed to a web endpoint. This is the reason why the last two attributes are required. This is, strictly speaking, not necessary for monitoring purposes since the deploying process might exist separately. The *Model* class also contains a number of *IO* instances representing the inputs of the ML model, also known as features, and an additional instance of *IO* representing the output. The assumption made is that the model accepts a fixed number of scalar values as input, and returns a single scalar value as output. This can also be generalized further.

C. IO

An *IO* instance represents an input or an output of the monitored ML model. It contains the name of the input/output and an enumerative attribute describing its data type. Depending on the underlying implementation, different data types could be supported or this information might even be unnecessary.

D. DriftDetector

A *DriftDetector* represents the periodic execution of an algorithm that indicates whether corrective action should be taken. The domain expert can define a *DriftDetector's* name and execution frequency. They can also define the algorithm that is to be executed. They can choose between a set of algorithms already included in the underlying platform or provide the source code for a custom one that is going to be integrated with the rest of the system. The specifics of this mechanism are further explained in the Generated Artifacts section.

In this version of the meta-model, a distinction has been made between two types of *DriftDetectors*. The first type requires a set of unlabelled samples that are received over time in addition to the set of labelled samples used to train the ML model. This type of *DriftDetector* is represented by the *DataDriftDetector* subclass. The second type requires a set of predicted labels produced by the ML model when provided with unlabelled samples for which the actual labels have been subsequently acquired. This type of *DriftDetector* is represented by the *ConceptDriftDetector* subclass. This distinction was made in order to simplify the underlying technical implementation of the system. Unifying the two types by designing one, more general class that is agnostic to the type of algorithm that is executed is planned as future work. Lastly, a *DriftDetector* also contains an email address attribute. In this initial version, sending email alerts is the only kind of action that can be taken in response to detected drift. Therefore, an email address attribute was added to *DriftDetector* instead of creating a separate class.

IV. Generated Artifacts

A MDE model that adheres to the meta-model presented in the previous section describes the behaviour of a ML monitoring system without specifying any technical implementation details. There are therefore multiple valid implementations, based on different technologies. One such implementation, based on Kubernetes[2], is presented below. Kubernetes was chosen because it is a vendor-neutral platform that can run on a wide variety of computing infrastructure, enabling experimentation. It is also widely used, which has resulted in the formation of a rich ecosystem of tools that extend its functionality. A number of these tools have been leveraged for this implementation.

Using the model-to-text (M2T) transformation language, EGL [9], a M2T transformation was implemented. The output of this transformation is a Kubernetes manifest which contains the description of the resources that need to be provisioned and the containerized applications that need to be executed on a cluster. The resulting system implements the behaviour specified by a domain expert in the MDE model provided. For some components of the system, open-source, custom Kubernetes resources are used. For the rest, containerized applications were developed and indexed in a container registry so that they can be referenced in the manifest. The main advantage of this approach is that the complexity of the M2T transformation is reduced as no procedural code is generated.

Figure 2 shows a diagram of the kind of system that is described by the generated Kubernetes manifests. Below are the descriptions of the components that make up the depicted system.

A. Inference Service

This component provides two functionalities. Firstly, it contains a model server which is responsible for responding to inference requests sent by various applications. Secondly, it contains a logger which produces events containing information about the model server's inbound requests and outbound responses. These events can be consumed by other web

[2]https://kubernetes.io

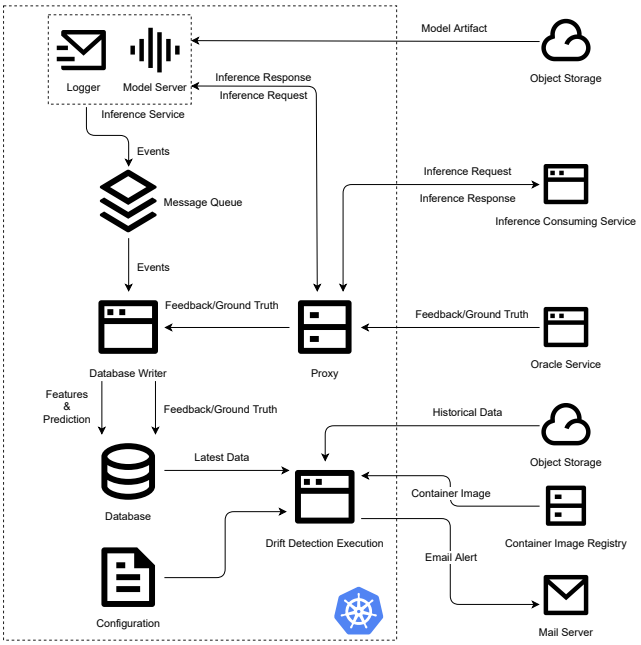

Fig. 2. Generated Artifacts

services. The Inference Service component is implemented as an *InferenceService* custom resource, which is developed by the Kubeflow[3] open-source project.

B. Message Queue

The Message Queue component is responsible for receiving events from the logger and routing them to the database writer service. This functionality is implemented using *Broker* and *Trigger* custom resources which are developed by the Knative[4] open-source project.

C. Database Writer

This component is responsible for initializing a database, based on the information provided by the domain expert, and storing the captured monitoring data in it. The first source of data, comes from consuming the events produced by the Inference Service's logger. These events contain the bodies of the requests/responses that are received/sent by the Inference Service's model server. Since the schema of the API employed by the model server is known, one can extract the values of the features and the prediction according to the domain expert's specification. The extracted values are subsequently stored in the previously initialized database. The second source of data, comes in the form of feedback about the ML model's performance. This is the ground truth labels for the inference requests previously sent to the Inference Service. Every inference request/response gets a unique ID attached to it so that it can later be matched with its corresponding ground truth information. The Database Writer is implemented using a *Service* custom resource by the Knative open-source project. The container that is executed by the Knative *Service* is a containerized web service developed for this specific task.

D. Database

This is a standard MySQL database used to persist the data as described above. It is implemented using *Deployment*, *Service* and *PersistentVolumeClaim* resources from Kubernetes' base API and official MySQL container images.

E. Drift Detection Execution

This component is responsible for the periodic execution of the drift detection algorithm specified by the domain expert. It is implemented as a *CronJob* resource from Kubernetes' base API which executes a container on a periodic basis. For this implementation, a containerized application was developed that retrieves test set data from the database and training set data from cloud storage and passes them to a function that implements the drift detection algorithm. The function returns a boolean that indicates whether drift was detected, along with a message. In the case of detected drift the message is included in an email alert sent to the relevant email address. If the domain expert wants to use their own custom algorithm, they can specify the container image that implements it. In general, they do not need to implement the parts responsible for data retrieval and notification sending. They only need to provide their custom algorithm implemented as a function with specific inputs and outputs so that it can be integrated with the base drift detection application template. More details of this functionality are provided in the supporting platform section.

F. Configuration

The containerized applications that were developed as part of this implementation, need to adapt their behaviour to match the domain expert's specification. This can be achieved by using configuration files. These are text files that are mounted to a container's file system. An application running inside the container can read them and adjust its behaviour accordingly. The configuration files are implemented as *ConfigMaps* from Kubernetes' base API.

G. Proxy

In order to serve inference requests and also receive feedback under the same host name a proxy is used. For every incoming HTTP request, the proxy checks the request's path. Depending on the path, it either forwards the request to the Inference service or the Database Writer. This functionality is implemented using a *VirtualService* custom resource developed by the Istio[5] open-source project.

V. Supporting Platform

It is technically possible for a domain expert to generate the Kubernetes manifests mentioned above by executing the M2T transformation on their workstation. They could then use a Kubernetes CLI tool to deploy to a cluster they have access to.

[3]https://kubeflow.org
[4]https://knative.dev
[5]https://istio.io

While possible, this process is not very user-friendly. Since the goal is to minimize the technical barriers that domain experts face when trying to deploy ML monitoring systems, a web application that simplifies this process is provided. Below, is an overview of the functionality offered by this web application.

The main functionality provided is the authoring and submission of MDE models. Users fill out a form whose fields correspond to the attributes of the various classes in the domain meta-model. When the user submits the form, the specified MDE model is serialized in Flexmi [10] format and sent to the application's back-end. In the back-end, the received MDE model is used as input for the model to text transformation described in the previous section. The resulting Kubernetes manifest is then used as input for Kubernetes' CLI tool which sends the objects specified in the manifest to the cluster's API server. This last step concludes the deployment of the ML monitoring system described by a domain expert in the form of a MDE model.

In addition to the main functionality, the user also has access to a number of auxiliary features. These are specific to the computing infrastructure used for this implementation and not designed with a focus on reusability . Despite that, they are developed in order to facilitate discussion with ML domain experts and can therefore contribute to the empirical evaluation of this work. The auxiliary features provided are the following:

- Listing all of the monitored ML models that are currently deployed. For each deployment two URLs are shown. One that can be used to send inference requests and a second one that can be used to send feedback/ground truth data.
- A form is provided for the upload of serialized ML models to cloud storage. In addition, the URLs of previously uploaded ML models are shown.
- A form is provided for the upload of files that contain training datasets to cloud storage. In addition, the URLs of previously uploaded training datasets are shown.
- Lastly, users can provide their own implementations of drift detection algorithms in the form of a Python function. Additionally, if their code imports any third-party packages, they need to provide a Python requirements file. In the back-end, the user-provided code is combined with the drift detector's base template and a container image is built. The image is pushed to a registry and its URI is listed in the user interface so it can be included in MDE models. This feature increases the solution's flexibility without requiring the user to know how the ML monitoring system is implemented.

VI. Future Work

This work is intended to be a proof of concept that showcases the feasibility of MDE techniques applied in the ML monitoring area. Below are some of the improvements that need to be made for a more comprehensive solution:

- The scheduling of drift detection executions needs to cover more complex scenarios. For example, in addition to executing periodically, one might want to define additional constraints, such as minimum amount of samples received between executions.
- There is a need for an abstraction that is expressive enough to describe any kind of drift detection algorithm. That would remove the need for making distinctions between drift types in the meta-model layer.
- The list of actions that can be taken in response to drift needs to be expanded. Also, it would be beneficial to provide a mechanism for describing complex scenarios in which a combinations of actions are prescribed when certain conditions are met.
- The ability to incrementally update a deployed ML monitoring system would be beneficial. This would allow domain experts to modify parts of their system without redeploying it in its entirety.

Once all of the needed improvements are made, a comprehensive evaluation of the improved solution shall commence. The solution could be evaluated on two different bases. Firstly, one could investigate whether the solution is general enough to describe a substantial number of monitoring techniques found in the literature. Secondly, an empirical evaluation could be carried out to answer whether the solution reduces the effort required for the deployment of ML monitoring systems.

Acknowledgment

The work in this paper has been partially supported by the Lowcomote project, that received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 813884.

References

[1] J. Friedman, T. Hastie, R. Tibshirani *et al.*, *The elements of statistical learning*. Springer series in statistics New York, 2001, vol. 1, no. 10.
[2] M. Mohri, A. Rostamizadeh, and A. Talwalkar, *Foundations of machine learning*. MIT press, 2018.
[3] J. C. Schlimmer and R. H. Granger, "Incremental learning from noisy data," *Mach. Learn.*, vol. 1, no. 3, pp. 317–354, 1986.
[4] B. Schölkopf, D. Janzing, J. Peters, E. Sgouritsa, K. Zhang, and J. Mooij, "On causal and anticausal learning," in *29th International Conference on Machine Learning (ICML 2012)*. International Machine Learning Society, 2012, pp. 1255–1262.
[5] M. Salganicoff, "Tolerating concept and sampling shift in lazy learning using prediction error context switching," *Artif. Intell. Rev.*, vol. 11, no. 1-5, pp. 133–155, 1997.
[6] A. Storkey, "When training and test sets are different: characterizing learning transfer," *Dataset shift in machine learning*, vol. 30, pp. 3–28, 2009.
[7] M. Kull and P. Flach, "Patterns of dataset shift," in *First International Workshop on Learning over Multiple Contexts (LMCE) at ECML-PKDD*, 2014.
[8] R. Elwell and R. Polikar, "Incremental learning of concept drift in nonstationary environments," *IEEE Trans. Neural Networks*, vol. 22, no. 10, pp. 1517–1531, 2011.
[9] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. Polack, "The epsilon generation language," in *European Conference on Model Driven Architecture-Foundations and Applications*. Springer, 2008, pp. 1–16.
[10] D. S. Kolovos and R. F. Paige, "Towards a modular and flexible human-usable textual syntax for emf models." in *MODELS Workshops*, 2018, pp. 223–232.