Challenges inLive Monitoring of Machine Learning Systems

Patrick Baier[1] and Stanimir Dragiev[2]

[1] Hochschule Karlsruhe – University of Applied Sciences
patrick.baier@h-ka.de
[2] Zalando Payments
stanimir.dragiev@zalando.de

Abstract. A machine learning (ML) system involves multiple layers of software and therefore needs monitoring to ensure areliable operation. As opposed to traditional software services, the quality of its predictions can only be guaranteed if the data that flows into the system follows a similar distribution as the data the ML model was trained on. This poses additional requirements on monitoring. In this paper we outline a scheme for monitoring ML services based on feature distribution comparison between the data used for training and for live prediction. To showcase this we introduce payment risk prediction as an application scenario. Its long feedback delays and real time requirements motivate monitoring and at the same time holds specific challenges which we address. In this context we discuss trade-offs for the practical implementation of the monitoring scheme andshare our best practices.

Keywords: reliable machine learning, monitoring, production systems, feature distribution, non-stationarity

1 Introduction

Live monitoring of software systems is an important and well studied field [1] that helps to prevent unexpected service interruption and ensures stability and reliability. Monitoring typically involves collecting metrics about a system and checking if these values lie within a range of expected values. If this is not the case, alerts are triggered to warn a system operator who checks for the healthiness of the system. Example metrics are the size of the free heap memory of a software process or the CPU utilization of the machine that it is running on. While monitoring of software systems has a long history and is widely applied, the proliferation of systems that rely on machine learning (ML) models brings a new challenge to this field.

A typical ML system consumes input data and maps it to a prediction, which is used in a downstream decision engine. For instance, a fraud detection system uses payment transaction data to predict if a transaction is fraud or eligible. This prediction is then used to decide if a warning to the card holder should be triggered or even to cancel the transactions. To ensure that the predictions of the ML system are accurate, the model is tested after training and before live deployment on a held-out test set with respect to typical quality metrics such as accuracy and area under the ROC curve. In general, the prediction quality of a ML model on unseen input data is within expectation only if the input data is similar to the training data [2]. Technically, similarity means that unseen data points are drawn from the same distribution as the data used for training the model. The assessed quality measure on the test set – e.g. accuracy – cannot be promised in

live operation if the live distribution and the training distribution differ. Hence, a live monitoring system is needed that periodically checks that the data which is served to the ML model in the live system is close enough to the training data distribution.

There are two main reasons why the input data distribution in a live system may differ from the training data distribution: The calculation of an input feature in some preceding system is flawed (e.g. money values are sent in euros instead of cents) or there is a natural data drift triggered by, for instance, phenomena like inflation. Without proper monitoring such data shifts can stay unnoticed. While the technical monitoring of the software stack can look perfectly fine, the quality of the predictions and hence the decisions based upon them may already suffer substantially. This happens long before the impact is measurable, resulting in big monetary damages. Thus, ML systems need an additional layer of monitoring that is concerned with checking for sane data distributions to meet the expected quality of service.

While several deployment related concerns of ML models are already tackled [3,4] the problems that arise with live monitoring of such systems are not well studied yet. In this paper we propose a ML monitoring system which is based on the experiences of running and monitoring ML systems in production environments for almost ten years. Besides giving some basic overview of the nuts and bolts of such a monitoring system, we highlight the technical challenges and trade-offs that arise with it, e.g. windowing, seasonality and computational trade-offs.

2 Basic Monitoring System

Before we discuss the technical challenges of ML monitoring, we start with abasic overview of the monitoring system we propose. To make this more tangible, we first introduce an example application scenario which will be used to lay out the subsequent concepts.

2.1 Application Scenario

Given is an online payment provider that handles the complexity of payment transactions on behalf of an online merchant. The provider strives for the best customer experience which includes seamless transaction processingand convenient payment options, e.g. credit/debit card, cash-on-delivery, etc., and most notably deferred payment. For each transaction, the provider needs to decide if a consumer can be offered a deferred payment option. For instance, if a person wants to buy shoes online, will the person have the option to pay only after receiving the shoes? In general, deferred payment options make the online buying experience closer to the offline shopping and thus increase conversion rates of online shops, but also bear the risk of not receiving the money from the customer (also known as *payment default*). To make sure that deferred payment options are offered only to the right customers, the payment provider needs to predict the risk of a payment default for every customer. Moreover, this predictionhas to be finished before the customer arrives at the payment selection page of its shopping session. To tackle this problem payment providers typically employ ML systems that use features about the customer and the current shopping session. The output is a prediction of the payment default likelihood of a customer for this purchase. Such a model can be trained on historic payment data and is widely employed among online payment providers.

This scenario has two special characteristics which make monitoring especially challenging: (1) The model has to provide its prediction in *real-time*, i.e. a payment default

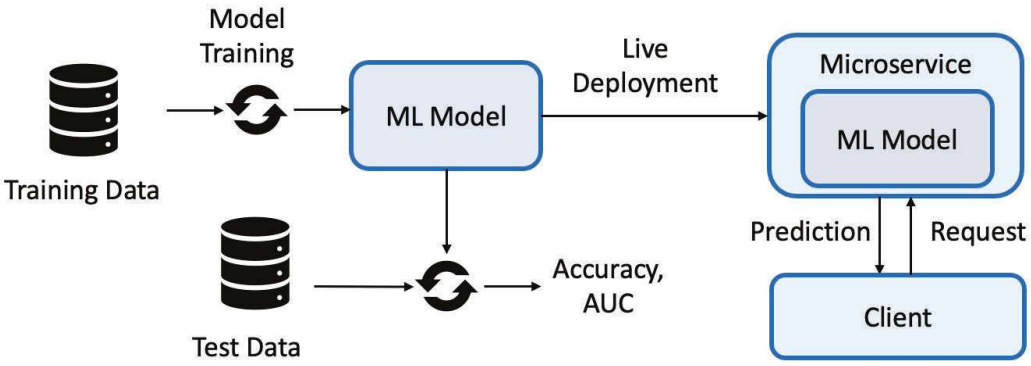


Fig. 1: Model Life Cycle

probability has to be provided in a matter of a few hundred milliseconds. Models with such a requirement are typically running encapsulated in web services that are deployed in the cloud. (2) The scenario has a *delayed feedback loop*. This means that we can only evaluate the decision of the model after a long delay. In the above scenario, if the model predicts that deferred payment should be granted to a customer it can take weeks until the money comes in. Only then a label for the data point (customer paid or defaulted) is available and we know if the model's prediction was right. Other examples for systems with such a delayed feedback loop are order return prediction or customer churn prediction.

Especially the combination of these two properties highlights the importance of monitoring: We needto ensure that the model is working as expected, otherwise we may only find about it weeks later when the quality of the system's decisions can finally be assessed. In worst case, several weeks of wrong predictions may result in a complete financial fiasco for a company.

2.2 Model Life Cycle

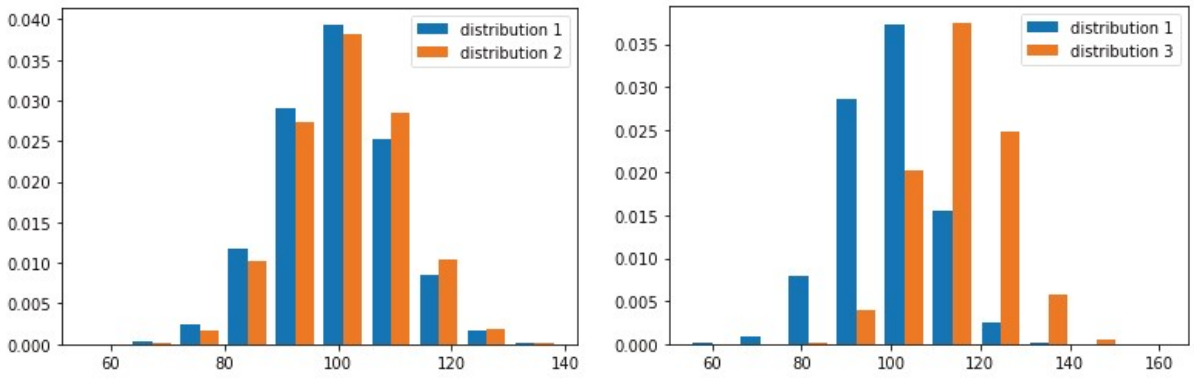
To make the contextual dependencies of model monitoring visible, we shortly sketch the typical ML workflow that is preceding the monitoring (see Figure 1). What is left aside here, are all the phases that precede model training (e.g. label definition, data acquisition, feature engineering, etc).

The model life cycle starts by training the model on the training data. To check the models performance it is evaluated on a hold-out test data set and performance metrics like accuracy or area under the curve are calculated. If everything looks fine, the model is deployed to the live system where it receives requests from a client application. A request contains a data point to predict on, i.e. it contains all features that the model needs for a prediction. The model returns its output probability to the client within a few hundred milliseconds. Typically only a fewrequests are routed at this stage to the newly deployed model in order to do a final check on its technical readiness. After that more and more requests are gradually routed to the new model until the full traffic arrives there. That is the point when model monitoring kicks in to make sure that the model runs reliably and that performance observed on the test data can be expected under real conditions.

2.3 Monitoring Metrics

The most important question in monitoring is *what* exactly to monitor to ensure that the system is working as expected. In systems with a delayed feedback loop we cannot just monitor a metric like accuracy, since a feedback about the model's decision is not immediately observable. However, as a proxy we can monitor the sanity of the data that is flowing into the model, which are the input feature values.

It is widely understood that ML models only perform as expected if the data fed to the live system is similar to the data used to train the model [2]. As a result, the primary candidate metric to monitor is the difference between the data distributions of the training data and the live data that is currently flowing into the model. Both are empirical data distributions that can be compared by statistical metrics like the Wasserstein distance (also known as Earth mover's distance) or the Kullback–Leibler divergence. Such a distance score quantifies the similarity of the two distributions. Figure 2 shows an example with different empirical distributions and the corresponding Wasserstein distance. Note that for the rest of this paper, we assume that the Wasserstein distance is used for comparing distributions.



(a) Similar distributions, WD=0.9 (b) Different distributions, WD=15

Fig. 2: Wasserstein distances (WD) when comparing two empirical distributions.

2.4 Monitoring System

The proposed monitoring system works as follows: Every t seconds the system computes for every input feature of the model its current live distribution considering the last n requests that were sent to the model. For every feature, this distribution is compared to the training data distribution using the Wasserstein distance. As a result, the monitoring outputs every t seconds the Wasserstein distance for every feature of the model. If one of the computed scores lies above a preselected alerting threshold the monitoring system triggers an alert to a system operator to investigates the cause for the distribution shift. We will quickly discuss possible reasons for this in the following subsection. The whole monitoring process in summarized in Figure 3 that shows an example flow for one feature.

Finding the right alerting threshold is crucial: If the chosen threshold is too low, false positive alerts are triggered, which means that the monitoring alerts even though the feature distributions did not significantly change. This can for instance happen due to

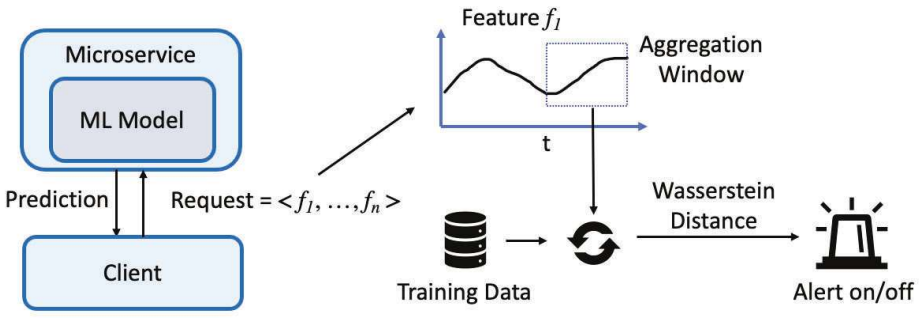


Fig. 3: Overview of monitoring process for one feature.

some recent outlier data points in the live system. On the other hand, if the alerting threshold is too high, real distribution changes can stay unnoticed since no alerts are triggered. Problems may thennot be detected fast enough and the whole purpose of monitoring becomes obsolete. A typical way to find a good threshold is too start with a rather low value and then slightly increase the threshold if the resulting alerts can safely be identified as false positives.

The description of the system above contains two parameters that must be set before the monitoring system can operate: (1) Time t that determines how often a distribution comparison is triggered. (2) Window size n which determines how many recent live requests are considered for calculating the live data distribution. Finding good parameter settings is again not straightforward but crucial. Both are discussed in more detail in the technical challenges in Section 3.

2.5 Sources of Errors

An alarm triggered by the monitoring system means that there is a change in the data distribution for at least one of the features. The cause of such a shift typically comes from one of the following sources: (1) There is a technical problem with delivering the correct feature value to the model, i.e. the client sends the wrong data. (2) The input feature data suffers from a natural distribution shift.

Inthe first case, the model receives wrong data from the preceding system. This can have several causes. For instance, a downtime in a database or a unstable network connection may lead to missing feature values which over time lead to a changed data distribution. Another problem could be a newly introduced bug in the preceding system that alters a feature value. For instance, money values are sent in cents instead of euros after a new software deployment. It is crucial to detect such cases since such a tiny bug can have immense effects on the output of a ML model.

In contrast, a natural shift in the input data does not stem from a technical problem but some underlying phenomena in the data itself. This can typically be observed by a gradual shift in the live data over time. The resolution to such a problem could be to analyse the data shift and remove any trend before the data goes into the model. Another alternative is to re-train the model in short time intervals to always include the freshest available data.

3 Technical Challenges

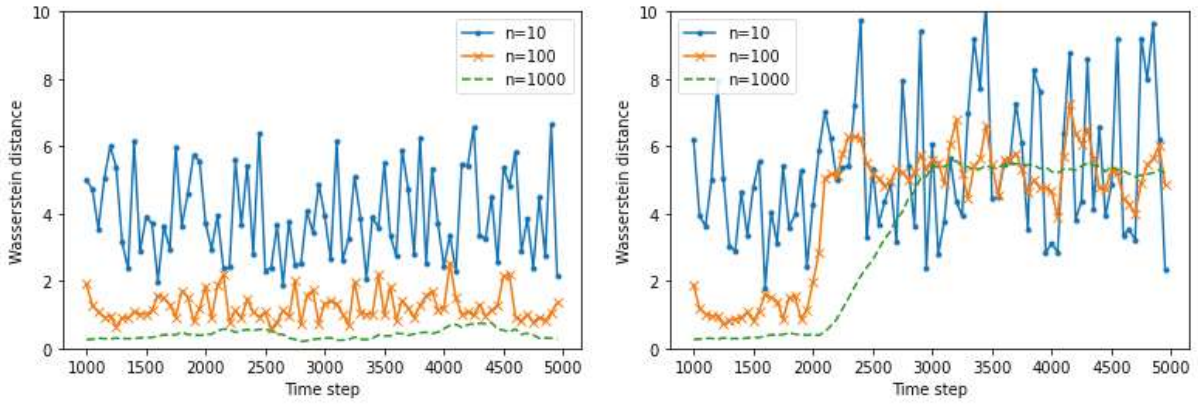
In this section we look into the technical challenges that are inherent to the presented monitoring system. To solve them, one has to decide for certain trade-offs that are specific to the available data and the application scenario.

3.1 Aggregation Window Size

One important parameter for the system is the size of the aggregation window which is used to determine the distribution of the live data. In the previous section this window size was denoted as parameter n. The choice of this parameter has trade-offs in both directions: If n is chosen rather small, there is only very recent data in the aggregation window. This is on the one hand desirable since we prefer to build the live data distribution from relatively fresh data. For instance, consider the extreme case in which the aggregation window contains all live data seen until time t. If there is a change ina feature at time $t+1$, it would take a long time to be visible in the live data distribution since the aggregation window is dominated by the old data.

On the other hand, choosing a small n could detect such distribution shifts very quickly but comes with problems regarding the representation of the distribution. Building the empirical distribution from only a small aggregation window suffers from uncertainty due to the small sample size. Only if we build the distribution from a large enough number of empirical data points, we can be sure to approximate the underlying distribution. Hence, if we choose n too small we will derive a wrong empirical distribution and the alerting system will kick in since the distance to the training data is too big.

To show this trade-offwe conducted a small experiment. For a sample feature, we created 10k training data points drawn from a Gaussian distribution with $\mu = 100$ and $\sigma^2 = 10$. Going back to the sample scenario, this feature could for instance represent the summed price of items in a customer order. To simulate live data we created a new data point at every time step which was drawn from the same distribution. In the two plots in Figure 4 at every 50 time steps the Wasserstein distance between the training and live distribution with window size $n \in \{10, 100, 1000\}$ is plotted.



(a) No data shift

(b) Data shift at $t = 2000$

Fig. 4: Effect of different window sizes on the Wasserstein distance

From Figure 4a we see that a small window size results in rather high Wasserstein distances even though the data comes from the same distribution. As discussed before,

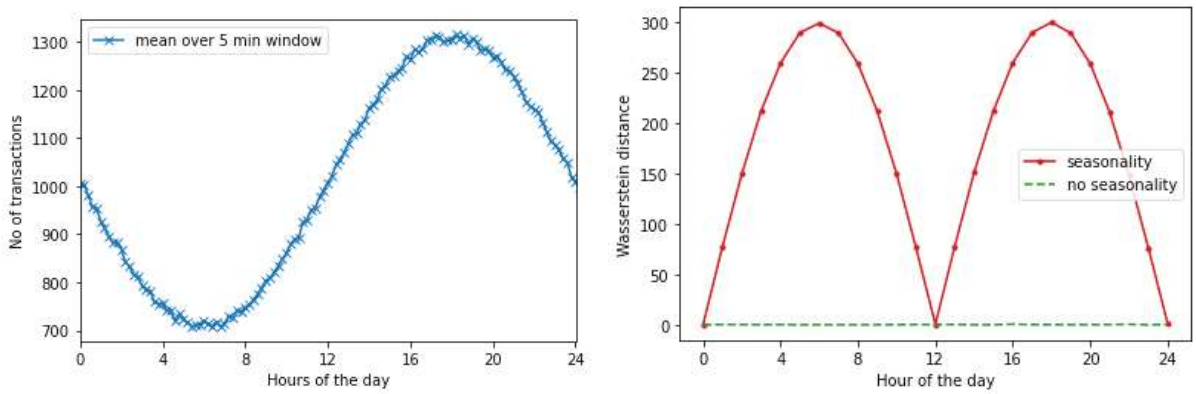
if the window size is chosen too small, the live distribution is not representative for the underlying distribution. Only if we increase n to 1000, the Wasserstein distance is small enough such that the two distributions can be considered equal. Hence, we have to choose n big enough to avoid false-positive alerts. In Figure 4b, we changed the distribution of the live data at time $t = 2000$ by reducing the mean of the Gaussian to 90. Here we can see that choosing a big n can lead to a rather late detection of a distribution shift, which may be very costly for the business.

To find a good windowsize in practice, we recommend to choose n as small as possible but at the same time to make sure that the Wasserstein distance stays low for data from the same distribution.

3.2 Seasonality

The second problem that can arise when monitoring distribution shifts is seasonality in input features. To illustrate this we assume that *the amount of payment transactions in the last hour* is a feature in the ML model of the aforementioned payment provider. The distribution of such a feature naturally fluctuates across one day. For an experiment we assumed the feature to fluctuate within one day as shown in Figure 5a. The curve in this figure shows at every time of the day the mean of all data points within the last five minutes. If we compare the live distribution of this feature against the distribution of the training data, we face the problem that the current live distribution contains a seasonal shift, while the training data is aggregated over all training data points and is static with $\mu = 1000$. As a result, the Wasserstein distance can increase significantly during the day even when using a big window size of $n = 1000$ (see the upper curve in Figure 5b).

To overcome this problem, we can limit the comparison of live data to only the set of training data that comes from the same time window. For instance, if we want to compare live vs. train distributionat 2pm, we compare the current live distributions only with training data points for which the data was also collected in that time frame, i.e. every day in the corresponding window before 2pm. In this way, the seasonality can be factored out and the distribution comparison results in a low Wasserstein scores (see the lower curve in Figure 5b).



(a) Feature with seasonality (b) Distribution Monitoring

Fig. 5: Effect of seasonality on distribution monitoring

3.3 Computational Aspects

One last consideration is about the computational effort that is required for model monitoring. The Wasserstein distance is computationally complex but has linear approximations [5] that are usually good enough for model monitoring.

Here we face again a trade-off: If the distribution difference is calculated frequently, computational costs increase and, hence, also increase monetary efforts for running the monitoring system. On the other hand, if the distribution difference is calculated only occasionally there can be a significant delay in detecting problems in the ML system. To find a goodtrade-off one has to consider the amount of traffic in the live system and determine how much computation is actually spend for calculating the Wasserstein distance. Based on this number a meaningful trade-off between computation costs and monitoring delay can be found.

4 RelatedWork

While monitoring in software systems is a field that is studied quite well, the additional complexitythat ML systems introduce to monitoring is only addressedso far infew publications. Breck et al. [3] were among the first that summarized the challenges that arise when running ML models in live systems. While theylist model monitoring as one important aspect, they do not go into technical details on how to implement this. Klaise et al. [6] discuss in their work aspects of monitoring and explainability of deployed models. In this context, they discuss statistics measure for detecting a drift in the live distribution but do not cover the comparison between live and training data distribution. The authors of this articles also provide an open source system that helps to automatically detect distribution drifts in the live system [7]. Finally, Paleyes et al. [8] review reports of deployed ML solutions and also shortly discuss the aspect of monitoring. However, none of the reported system has a delayed feedback loop and hence lacks a concept of distribution comparison which is necessary in this case.

5 Discussion

5.1 Monitoring adaptive systems

A questionthat may arise in Section 2.3 is why, in the first place, we should monitor changing feature distributions instead of building a model that can adapt to these changes? Indeed, some models can accommodate drifts. If a drift is predictable, it can be modeled explicitly. Unexpected drifts, on the other hand, can be accounted for by updating the model by online training on the newdata points. This is only possible by limiting the "memory" in the training process. Training data which does not look far in the past makes a model more sensible to recent changes. The smaller the training data horizon back in time, the faster the new models will pick up the new situation. However, a short training data period poses the risk that the new model "forgets" patterns available in the old data but not in the recent data. This introduces another trade-off for howto deal with drifts and is rather an argument for monitoring than an alternative: each setting favours a particular case and imposes a risk which is better discovered sooner than later.

5.2 Limitations

The approach described in this paper is based on monitoring individual features to catch a change in order to maintain a promised quality of the MLsystem. Examples can be constructed, however, that show a distribution change in a higher-dimensional feature space while the univariate distributions of the individual features remainstable. The MLsystem may not generalize as well as expected to some examples from the changed distribution and the performance can degrade. With the current approach, such a change may stay unnoticed. A remedy would be to extend the distribution comparisons to pairs, triples or higher tuples of features or even to the whole feature space. However, enumerating all such tuples introduces a huge overhead for a questionable benefit: knowing that, say, 20 features collectively deviate is not very actionable in general. In special cases, though, the collective behaviour of subsets of features may well be of interest.

While monitoring distributions does a good job to uncover changes, there are a class of problems which are rooted in the opposite. A popular write-up with practical advices for ML engineers [9] features "stale tables" as a particular pitfall which occurs more for ML systems than for others. Let us assume a table owned by another team has not been updated for a while. If a feature aggregates some counts from the table and these counts are missing for the recent past, the feature will slightly drift towards zero. For example, consider a feature that counts how often a customer visited a page in the past week. If a table with the dailyvisits of customers per page freezes, the aggregations will decrease. Sooner or later this will appear in the distribution comparison. However, if a feature does not slide a window, the distribution will not change. Here should be noted that monitoring feature distributions may help discover bugs opaque to traditional monitoring, however it mainly aims at detecting shifts innate to the environment.

6 Conclusion

The monitoring of an ML service is a prerequisite for the reliable operation and for maintaining service levels promised at the time of development of the ML models. Especially in a complex environment like payment risk prediction, without ML monitoring the enterprise is put at risk. Against the background of delayed feedback, not fully observable decision effects and non-stationary features, we outline the main traits of a monitoring system based on comparing the observed feature distributions. We discuss ways to aggregate and compare the distributions avoiding misalignment by making trade-offs needed in the practical implementation.

While monitoring ML services can discover a range of internal and external hazards, it is not a panacea, in particular it is not a substitute to the traditional software service monitoring; it is rather an augmentation to the existing well maintained monitoring practices.

References

1. Gao, L., Lu, M., Li, L., Pan, C.: A survey of software runtime monitoring. (2017)
2. Koh, P.W., Sagawa, S., Marklund, H., Xie, S.M., Zhang, M., Balsubramani, A., Hu, W., Yasunaga, M., Phillips, R.L., Gao, I., Lee, T., David, E., Stavness, I., Guo, W., Earnshaw, B., Haque, I., Beery, S.M., Leskovec, J., Kundaje, A., Pierson, E., Levine, S., Finn, C., Liang, P.: Wilds: A benchmark of in-the-wild distribution shifts. (2021)
3. Breck, E., Cai, S., Nielsen, E., Salib, M., Sculley, D.: What's your ML test score? A rubric for ML production systems. (2016)

4. Murphy, C., Kaiser, G., Arias, M.: An approach to software testing of machine learning applications. (2007)
5. Atasu, K., Mittelholzer, T.: Linear-complexity data-parallel earth mover's distance approximations. (2019)
6. Klaise, J., Looveren, A.V., Cox, C., Vacanti, G., Coca, A.: Monitoring and explainability of models in production (2020)
7. Van Looveren, A., Vacanti, G., Klaise, J., Coca, A., Cobb, O.: Alibi detect: Algorithms for outlier, adversarial and drift detection (2019)
8. Paleyes, A., Urma, R.G., Lawrence, N.D.: Challenges in deploying machine learning: a survey of case studies (2021)
9. Zinkevich, M.: Rules of machine learning: Best practices for ML engineering. https://developers.google.com/machine-learning/guides/rules-of-ml (2017)