What's your ML Test Score? A rubric for ML production systems

Eric Breck, Shanqing Cai, Eric Nielsen, Michael Salib, D. Sculley
Google, Inc.
{ebreck, cais, nielsene, msalib, dsculley}@google.com

Abstract

Using machine learning in real-world production systems is complicated by a host of issues not found in small toy examples or even large offline research experiments. Testing and monitoring are key considerations for assessing the production-readiness of an ML system. But how much testing and monitoring is enough? We present an ML Test Score rubric based on a set of actionable tests to help quantify these issues.

1 Introduction

Using machine learning in real-world software systems is complicated by a host of issues not found in small toy examples or even large offline experiments [1]. Based on years of prior experience using ML at Google, in systems such as ad click prediction [2] and the Sibyl ML platform [3], we have developed a set of best practices for using machine learning systems. We present these practices as a set of actionable tests, and offer a scoring system to measure how ready for production a given machine learning system is.

This rubric is intended to cover a range from a team just starting out with machine learning up through tests that even a well-established team may find difficult. We feel that presenting the entire list is useful to gauge a team's readiness to field a real-world ML system.

This document focuses on issues specific to building a system using ML, and so does not include generic software engineering best practices, such as good unit test coverage throughout and a well-defined binary release process for training and serving. Such strategies remain necessary as well. We do call out a few specific areas for unit or integration tests that have unique ML-related behavior.

Computing an ML Test Score. For each test, one point is awarded for executing the test manually and documenting and distributing the results. A second point is awarded if there is a system in place to run that test automatically on a repeated basis. The final ML Test Score is computed by taking the *minimum* of the scores aggregated for each of the 4 sections below.

- **0 points:** More of a research project than a productionized system.
- **1-2 points:** Not totally untested, but it is worth considering the possibility of serious holes in reliability.
- **3-4 points:** There's been first pass at basic productionization, but additional investment may be needed.
- **5-6 points:** Reasonably tested, but it's possible that more of those tests and procedures may be automated.
- **7-10 points:** Strong levels of automated testing and monitoring, appropriate for mission-critical systems.
- **12+ points:** Exceptional levels of automated testing and monitoring.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

System architecture. Here we assume a system architecture that extracts features from raw input data, feeds them into a training system that produces a model, which is then read into a serving system to make inferences and affect the user-facing behavior of the system. We also assume typical software engineering practice such as a source repository, continuous automated test framework, and an ability to conduct experiments comparing different system versions.

Related work. Software testing is well studied, as is machine learning, but their intersection has been less well explored in the literature. [4] review testing for scientific software more generally, and cite a number of articles such as [5], who present an approach for testing ML algorithms. These ideas are a useful complement for the tests we present, which are focused on testing the use of ML in a production system rather than just the correctness of the ML algorithm per se.

2 Tests for Features and Data

Machine learning systems differ from traditional software-based systems in that the behavior of ML systems is not specified directly in code but is learned from data. Therefore, while traditional software can rely on unit tests and integration tests of the code, here we attempt to add a sufficient set of *tests of the data*.

Test that the distributions of each feature match your expectations. One example might be to test that Feature A takes on values 1 to 5, or that the two most common values of Feature B are "Harry" and "Potter" and they account for 10% of all values. This test can fail due to real external changes, which may require changes in your model.

Test the relationship between each feature and the target, and the pairwise correlations between individual signals. It is important to have a thorough understanding of the individual features used in a given model; this is a minimal set of tests, more exploration may be needed to develop a full understanding. These tests may be run by computing correlation coefficients, by training models with one or two features, or by training a set of models that each have one of k features individually removed.

Test the cost of each feature. The costs of a feature may include added inference latency and RAM usage, more upstream data dependencies, and additional expected instability incurred by relying on that feature. Consider whether this cost is worth paying when traded off against the provided improvement in model quality.

Test that a model does not contain any features that have been manually determined as unsuitable for use. A feature might be unsuitable when it's been discovered to be unreliable, overly expensive, *etc.* Tests are needed to ensure that such features are not accidentally included (e.g. via copy-paste) into new models.

Test that your system maintains privacy controls across its entire data pipeline. While strict access control is typically maintained on raw data, ML systems often export and transform that data during training. Test to ensure that access control is appropriately restricted across the entire pipeline.

Test the calendar time needed to develop and add a new feature to the production model. The faster a team can go from a feature idea to it running in production, the faster it can both improve the system and respond to external changes.

Test all code that creates input features, both in training and serving. It can be tempting to believe feature creation code is simple enough to not need unit tests, but this code is crucial for correct behavior and so its continued quality is vital.

3 Tests for Model Development

While the field of software engineering has developed a full range of best practices for developing reliable software systems, the set of standards and practices for developing ML models in a rigorous fashion is still developing. It can be all too tempting to rely on a single-number summary metric to judge performance, perhaps masking subtle areas of unreliability. Careful testing is needed to search for potential lurking issues.

Test that every model specification undergoes a code review and is checked in to a repository. It can be tempting to avoid, but disciplined code review remains an excellent method for avoiding silly errors and for enabling more efficient incident response and debugging.

Test the relationship between offline proxy metrics and the actual impact metrics. For example, how does a one-percent improvement in accuracy or AUC translate into effects on metrics of user satisfaction, such as click through rates? This can be measured in a small scale A/B experiment using an intentionally degraded model.

Test the impact of each tunable hyperparameter. Methods such as a grid search [6] or a more sophisticated hyperparameter search strategy [7] not only improve predictive performance, but also can uncover hidden reliability issues. For example, it can be surprising to observe the impact of massive increases in data parallelism on model accuracy.

Test the effect of model staleness. If predictions are based on a model trained yesterday versus last week versus last year, what is the impact on the live metrics of interest? All models need to be updated eventually to account for changes in the external world; a careful assessment is important to guide such decisions.

Test against a simpler model as a baseline. Regularly testing against a very simple baseline model, such as a linear model with very few features, is an effective strategy both for confirming the functionality of the larger pipeline and for helping to assess the cost to benefit tradeoffs of more sophisticated techniques.

Test model quality on important data slices. Slicing a data set along certain dimensions of interest provides fine-grained understanding of model performance. For example, important slices might be users by country or movies by genre. Examining sliced data avoids having fine-grained performance issues masked by a global summary metric.

Test the model for implicit bias. This may be viewed as an extension of examining important data slices, and may reveal issues that can be root-caused and addressed. For example, implicit bias might be induced by a lack of sufficient diversity in the training data.

4 Tests for ML Infrastructure

An ML system often relies on a complex pipeline rather than a single running binary.

Test the reproducibility of training. Train two models on the same data, and observe any differences in aggregate metrics, sliced metrics, or example-by-example predictions. Large differences due to non-determinism can exacerbate debugging and troubleshooting.

Unit test model specification code. Although model specifications may seem like "configuration", such files can have bugs and need to be tested. Useful assertions include testing that training results in decreased loss and that a model can restore from a checkpoint after a mid-training job crash.

Integration test the full ML pipeline. A good integration test runs all the way from original data sources, through feature creation, to training, and to serving. An integration test should run both continuously as well as with new releases of models or servers, in order to catch problems well before they reach production.

Test model quality before attempting to serve it. Useful tests include testing against data with known correct outputs and validating the aggregate quality, as well as comparing predictions to a previous version of the model.

Test that a single example or training batch can be sent to the model, and changes to internal state can be observed from training through to prediction. Observing internal state on small amounts of data is a useful debugging strategy for issues like numerical instability.

Test models via a canary process before they enter production serving environments. Modeling code can change more frequently than serving code, so there is a danger that an older serving system will not be able to serve a model trained from newer code. This includes testing that a model can be loaded into the production serving binaries and perform inference on production input data at all. It also includes a canary process, in which a new version is tested on a small trickle of live data.

Test how quickly and safely a model can be rolled back to a previous serving version. A model "roll back" procedure is useful in cases where upstream issues might result in unexpected changes to model quality. Being able to quickly revert to a previous known-good state is as crucial with ML models as with any other aspect of a serving system.

5 Monitoring Tests for ML

Monitoring is crucial for models that automatically incorporate new data in a continual or ongoing fashion at training time, and is always needed for models that serve predictions in an on-demand fashion.

Test for upstream instability in features, both in training and serving. Upstream instability can create problems both at training and serving (inference) time. Training time instability is especially problematic when models are updated or retrained frequently. Serving time instability can occur even when the models themselves remain static. As examples, what alert would fire if one datacenter stops sending data? What if an upstream signal provider did a major version upgrade?

Test that data invariants hold in training and serving inputs. For example, test if Feature A and Feature B should always have the same number of non-zero values in each example, or that Feature C is always in the range $(0, 100)$ or that class distribution is about 10:1.

Test that your training and serving features compute the same values. The codepaths that actually generate input features may differ for training and inference time, due to tradeoffs for flexibility *vs.* efficiency and other concerns. This is sometimes called "training/serving skew" and requires careful monitoring to detect and avoid.

Test for model staleness. For models that continually update, this means monitoring staleness throughout the training pipeline, to be able to determine in the case of a stale model where the pipeline has stalled. For example, if a daily job stopped generating an important table, what alert would fire?

Test for NaNs or infinities appearing in your model during training or serving. Invalid numeric values can easily crop up in your learning model, and knowing that they have occurred can speed diagnosis of the problem.

Test for dramatic or slow-leak regressions in training speed, serving latency, throughput, or RAM usage. The computational performance (as opposed to predictive quality) of an ML system is often a key concern at scale, and should be monitored via specialized regression testing. Dramatic regressions and slow regressions over time may require different kinds of monitoring.

Test for regressions in prediction quality on served data. For many systems, monitoring for nonzero bias can be an effective canary for identifying real problems, though it may also result from changes in the world.

Acknowledgments

We are very grateful to Keith Arner, Gary Holt, Todd Phillips, Todd Underwood, and Cory Williams for many helpful discussions and comments on drafts of this paper, as well as for sharing their experiences that led to the ideas here.

References

[1] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[2] H. Brendan McMahan, Gary Holt, D. Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, Sharat Chikkerur, Dan Liu, Martin Wattenberg, Arnar Mar Hrafnkelsson, Tom Boulos, and Jeremy Kubica. Ad click prediction: A view from the trenches. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1222–1230, New York, NY, USA, 2013. ACM.

[3] Tushar Chandra, Eugene Ie, Kenneth Goldman, Tomas Lloret Llinares, Jim McFadden, Fernando Pereira, Joshua Redstone, Tal Shaked, and Yoram Singer. Sibyl: a system for large scale machine learning. *Keynote I PowerPoint presentation, Jul*, 28, 2010.

[4] Upulee Kanewala and James M Bieman. Testing scientific software: A systematic literature review. *Information and software technology*, 56(10):1219–1232, 2014.

[5] Chris Murphy, Gail E Kaiser, and Marta Arias. An approach to software testing of machine learning applications. In *SEKE*, page 167. Citeseer, 2007.

[6] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. 2003.

[7] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. In *Advances in neural information processing systems*, 2012.