The ML Test Score: A Rubric for ML Production Readiness and Technical Debt Reduction

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

***Abstract*—Creating reliable, production-level machine learning systems brings on a host of concerns not found in small toy examples or even large offline research experiments. Testing and monitoring are key considerations for ensuring the production-readiness of an ML system, and for reducing technical debt of ML systems. But it can be difficult to formulate specific tests, given that the actual prediction behavior of any given model is difficult to specify *a priori*. In this paper, we present 28 specific tests and monitoring needs, drawn from experience with a wide range of production ML systems to help quantify these issues and present an easy to follow road-map to improve production readiness and pay down ML technical debt.**

***Keywords*-Machine Learning, Testing, Monitoring, Reliability, Best Practices, Technical Debt**

I. Introduction

As machine learning (ML) systems continue to take on ever more central roles in real-world production settings, the issue of *ML reliability* has become increasingly critical. ML reliability involves a host of issues not found in small toy examples or even large offline experiments, which can lead to surprisingly large amounts of technical debt [1]. Testing and monitoring are important strategies for improving reliability, reducing technical debt, and lowering long-term maintenance cost. However, as suggested by Figure 1, ML system testing is also more complex a challenge than testing manually coded systems, due to the fact that ML system behavior depends strongly on data and models that cannot be strongly specified *a priori*. One way to see this is to consider ML training as analogous to compilation, where the source is both code and training data. By that analogy, training data needs testing like code, and a trained ML model needs production practices like a binary does, such as debuggability, rollbacks and monitoring.

So, what should be tested and how much is enough? In this paper, we try to answer this question with a *test rubric*, which is based on engineering decades of production-level ML systems at Google, in systems such as ad click prediction [2] and the Sibyl ML platform [3].

We present a rubric as a set of 28 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. Note that this rubric focuses on issues specific to ML systems, and so does not include generic software engineering best practices such as ensuring good unit test coverage and a well-defined binary release process. 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.

How to read the tests: Each test is written as an assertion; our recommendation is to test that the assertion is true, the more frequently the better, and to fix the system if the assertion is not true.

Doesn't this all go without saying?: Before we enumerate our suggested tests, we should address one objection the reader may have – obviously one should write tests for an engineering project! While this is true in principle, in a survey of several dozen teams at Google, *none* of these tests was implemented by more than 80% of teams (though, even in a engineering culture valuing rigorous testing, many of these ML-centric tests are non-obvious). Conversely, most tests had a nonzero score for at least half of the teams surveyed; our tests do represent practices that teams find to be worth doing.

In this paper, we are largely concerned with supervised ML systems that are trained continuously online and perform rapid, low-latency inference on a server. Features are often derived from large amounts of data such as streaming logs of incoming data. However, most of our recommendations apply to other forms of ML systems, such as infrequently trained models pushed to client-side systems for inference.

A. Related work

Software testing is well studied, as is machine learning, but their intersection has been less well explored in the literature. [4] reviews testing for scientific software more generally, and cites 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.

Zinkevich provides extensive advice on building effective machine learning models in real world systems [6]. Those rules are complementary to this rubric, which is more

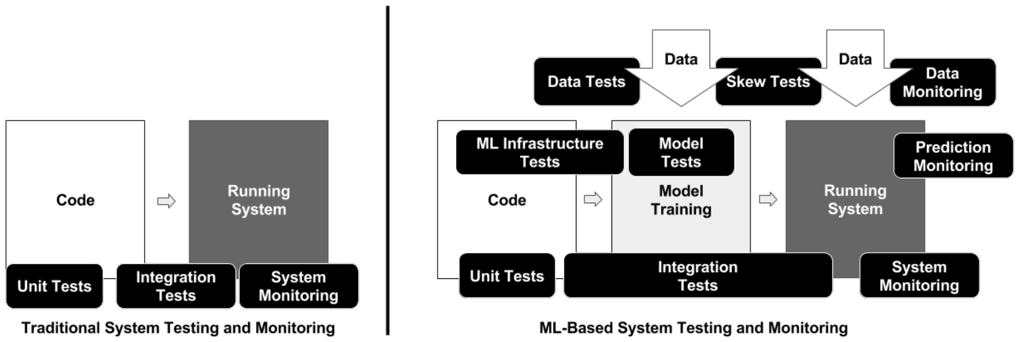


Figure 1. **ML Systems Require Extensive Testing and Monitoring.** The key consideration is that unlike a manually coded system (left), ML-based system behavior is not easily specified in advance. This behavior depends on dynamic qualities of the data, and on various model configuration choices.

concerned with determining how reliable an ML system is rather than how to build one.

Issues of surprising sources of technical debt in ML systems has been studied before [1]. It has been noted that the prior work has identified problems but been largely silent on how to address them; this paper details actionable advice drawn from practice and verified with extensive interviews with the maintainers of 36 real world systems.

II. 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*.

Data 1: Feature expectations are captured in a schema: It is useful to encode intuitions about the data in a *schema* so they can be automatically checked. For example, an adult human is surely between one and ten feet in height. The most common word in English text is probably 'the', with other word frequencies following a power-law distribution. Such expectations can be used for tests on input data during training and serving (see test Monitor 2).

How? To construct the schema, one approach is to start with calculating statistics from training data, and then adjusting them as appropriate based on domain knowledge. It may also be useful to start by writing down expectations and then compare them to the data to avoid an anchoring bias. Visualization tools such as Facets[1] can be very useful for analyzing the data to produce the schema. Invariants to capture in a schema can also be inferred automatically from your system's behavior [8].

1	Feature expectations are captured in a schema.
2	All features are beneficial.
3	No feature's cost is too much.
4	Features adhere to meta-level requirements.
5	The data pipeline has appropriate privacy controls.
6	New features can be added quickly.
7	All input feature code is tested.

Table I
Brief listing of the seven Data Tests.

Data 2: All features are beneficial: A kitchen-sink approach to features can be tempting, but every feature added has a software engineering cost. Hence, it's important to understand the value each feature provides in additional predictive power (independent of other features).

How? Some ways to run this test are 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.

Data 3: No feature's cost is too much: It is not only a waste of computing resources, but also an ongoing maintenance burden to include ϵ-features that add only minimal predictive benefit [1].

How? To measure the costs of a feature, consider not only added inference latency and RAM usage, but also more upstream data dependencies, and additional expected instability incurred by relying on that feature. See Rule#22 [6] for further discussion.

Data 4: Features adhere to meta-level requirements: Your project may impose requirements on the data coming in to the system. It might prohibit features derived from user data, prohibit the use of specific features like age, or simply prohibit any feature that is deprecated. It might require all features be available from a single source. However, during model development and experimentation, it is typical to try out a wide variety of potential features to improve prediction quality.

How? Programmatically enforce these requirements, so that all models in production properly adhere to them.

Data 5: The data pipeline has appropriate privacy controls: Training data, validation data, and vocabulary files all have the potential to contain sensitive user data. While teams often are aware of the need to remove personally identifiable information (PII), during this type of exporting and

[1]https://pair-code.github.io/facets/

transformations, programming errors and system changes can lead to inadvertent PII leakages that may have serious consequences.

How? Make sure to budget sufficient time during new feature development that depends on sensitive data to allow for proper handling. Test that access to pipeline data is controlled as tightly as the access to raw user data, especially for data sources that haven't previously been used in ML. Finally, test that any user-requested data deletion propagates to the data in the ML training pipeline, and to any learned models.

Data 6: New features can be added quickly: The faster a team can go from a feature idea to the feature running in production, the faster it can both improve the system and respond to external changes. For highly efficient teams, this can be as little as one to two months even for global-scale, high-traffic ML systems. Note that this can be in tension with Data 5, but privacy should always take precedence.

Data 7: All input feature code is tested: Feature creation code may appear simple enough to not need unit tests, but this code is crucial for correct behavior and so its continued quality is vital. Bugs in features may be almost impossible to detect once they have entered the data generation process, especially if they are represented in both training and test data.

III. Tests for Model Development

While the field of software engineering has developed a full range of best practices for developing reliable software systems, similar best-practices for ML model development are still emerging.

Model 1: Every model specification undergoes a code review and is checked in to a repository: It can be tempting to avoid code review out of expediency, and run experiments based on one's own personal modifications. In addition, when responding to production incidents, it's crucial to know the exact code that was run to produce a given learned model. For example, a responder might need to re-run training with corrected input data, or compare the result of a particular modification. Proper version control of the model specification can help make training auditable and improve reproducibility.

1	Model specs are reviewed and submitted.
2	Offline and online metrics correlate.
3	All hyperparameters have been tuned.
4	The impact of model staleness is known.
5	A simpler model is not better.
6	Model quality is sufficient on important data slices.
7	The model is tested for considerations of inclusion.

Table II
BRIEF LISTING OF THE SEVEN MODEL TESTS

Model 2: Offline proxy metrics correlate with actual online impact metrics: A user-facing production system's impact is judged by metrics of engagement, user happiness, revenue, and so forth. A machine learning system is trained to optimize loss metrics such as log-loss or squared error. A strong understanding of the relationship between these offline proxy metrics and the actual impact metrics is needed to ensure that a better scoring model will result in a better production system.

How? The offline/online metric relationship can be measured in one or more small scale A/B experiments using an intentionally degraded model.

Model 3: All hyperparameters have been tuned: A ML model can often have multiple hyperparameters, such as learning rates, number of layers, layer sizes and regularization coefficients. Choice of the hyperparameter values can have dramatic impact on prediction quality.

How? Methods such as a grid search [9] or a more sophisticated hyperparameter search strategy [10] [11] not only improve prediction quality, but also can uncover hidden reliability issues. Substantial performance improvements have been realized in many ML systems through use of an internal hyperparameter tuning service[12][2].

Model 4: The impact of model staleness is known: Many production ML systems encounter rapidly changing, non-stationary data. Examples include content recommendation systems and financial ML applications. For such systems, if the pipeline fails to train and deploy sufficiently up-to-date models, we say the model is *stale*. Understanding how model staleness affects the quality of predictions is necessary to determine how frequently to update the model. 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? Most models need to be updated eventually to account for changes in the external world; a careful assessment is important to decide how often to perform the updates (see Rule 8 in [6] for related discussion).

How? One way of testing the impact of staleness is with a small A/B experiment with older models. Testing a range of ages can provide an age-versus-quality curve to help understand what amount of staleness is tolerable.

Model 5: A simpler model is not better: 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.

Model 6: Model quality is sufficient on all important data slices: Slicing a data set along certain dimensions of interest can improve fine-grained understanding of model quality. Slices should distinguish subsets of the data that might behave qualitatively differently, for example, users by

[2]The service is closely related to HyperTune[13].

Table III
BRIEF LISTING OF THE ML INFRASTRUCTURE TESTS

1	Training is reproducible.
2	Model specs are unit tested.
3	The ML pipeline is Integration tested.
4	Model quality is validated before serving.
5	The model is debuggable.
6	Models are canaried before serving.
7	Serving models can be rolled back.

country, users by frequency of use, or movies by genre. Examining sliced data avoids having fine-grained quality issues masked by a global summary metric, e.g. global accuracy improved by 1% but accuracy for one country dropped by 50%. This class of problems often arises from a fault in the collection of training data, that caused an important set of training data to be lost or late.

How? Consider including these tests in your release process, e.g. release tests for models can impose absolute thresholds (e.g., error for slice x must be <5%), to catch large drops in quality, as well as incremental (e.g. the change in error for slice x must be <1% compared to the previously released model).

Model 7: The model has been tested for considerations of inclusion: There have been a number of recent studies on the issue of *ML Fairness* [14], [15], which may arise inadvertently due to factors such as choice of training data. For example, Bolukbasi *et al.* found that a word embedding trained on news articles had learned some striking associations between gender and occupation that may have reflected the content of the news articles but which may have been inappropriate for use in a predictive modeling context [14]. This form of potentially overlooked biases in training data sets may then influence the larger system behavior.

How? Diagnosing such issues is an important step for creating robust modeling systems that serve all users well. Tests that can be run include examining input features to determine if they correlate strongly with protected user categories, and slicing predictions to determine if prediction outputs differ materially when conditioned on different user groups.

Bolukbasi et al. [14] propose one method for ameliorating such effects by projecting embeddings to spaces that collapse differences along certain protected dimensions. Hardt et al propose a post-processing step in model creation to minimize disproportionate loss for certain groups in the manner of [15]. Finally, the approach of collecting more data to ensure data representation for potentially under-represented categories or subgroups can be effective in many cases.

IV. TESTS FOR ML INFRASTRUCTURE

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

Infra 1: Training is reproducible: Ideally, training twice on the same data should produce two identical models. Deterministic training dramatically simplifies reasoning about the whole system and can aid auditability and debugging. For example, optimizing feature generation code is a delicate process but verifying that the old and new feature generation code will train to an identical model can provide more confidence that the refactoring was correct. This sort of diff-testing relies entirely on deterministic training.

Unfortunately, model training is often not reproducible in practice, especially when working with non-convex methods such as deep learning or even random forests. This can manifest as a change in aggregate metrics across an entire dataset, or, even if the aggregate performance appears the same from run to run, as changes on individual examples.

Random number generation is an obvious source of non-determinism, which can be alleviated with seeding. But even with proper seeding, initialization order can be underspecified so that different portions of the model will be initialized at different times on different runs leading to non-determinism. Furthermore, even when initialization is fully deterministic, multiple threads of execution on a single machine or across a distributed system [16] may be subject to unpredictable orderings of training data, which is another source of non-determinism.

How? Besides working to remove nondeterminism as discussed above, ensembling models can help.

Infra 2: Model specification code is unit tested: Although model specifications may seem like "configuration", such files can have bugs and need to be tested. Unfortunately, testing a model specification can be very hard. Unit tests should run quickly and require no external dependencies but model training is often a very slow process that involves pulling in lots of data from many sources.

How? It's useful to distinguish two kinds of model tests: tests of API usage and tests of algorithmic correctness. We plan to release an open source framework implementing some of these tests soon.

ML APIs can be complex, and code using them can be wrong in subtle ways. Even if code errors would be apparent after training (due to a model that fails to train or results in poor performance), training is expensive and so the development loop is slow. We have found in practice that a simple unit test to generate random input data, and train the model for a single step of gradient descent is quite powerful for detecting a host of common library mistakes, resulting in a much faster development cycle. Another useful assertion is that a model can restore from a checkpoint after a mid-training job crash.

Testing correctness of a novel implementation of an ML algorithm is more difficult, but still necessary – it is not sufficient that code produces a model with high quality predictions, but that it does so for the expected reasons. One solution is to make assertions that specific subcomputations of the algorithm are correct, e.g. that a specific part of an RNN was executed exactly once per element of the input sequence. Another solution involves not training to completion in the unit test but only training for a few iterations and verifying that loss decreases with training. Still another is to purposefully train a model for overfitting: if one can get a model to effectively memorize its training data, then that provides some confidence that learning reliably happens. When testing models, pains should be taken to avoid "golden tests", i.e., tests that partially train a model and compare the results to a previously generated model – such tests are difficult to maintain over time without blindly updating the golden file. In addition to problems in training non-determinism, when these tests do break they provide very little insight into how or why. Additionally, flaky tests remain a real danger here.

Infra 3: The full ML pipeline is integration tested: A complete ML pipeline typically consists of assembling training data, feature generation, model training, model verification, and deployment to a serving system. Although a single engineering team may be focused on a small part of the process, each stage can introduce errors that may affect subsequent stages, possibly even several stages away. That means there must be a fully automated test that runs regularly and exercises the entire pipeline, validating that data and code can successfully move through each stage and that the resulting model performs well.

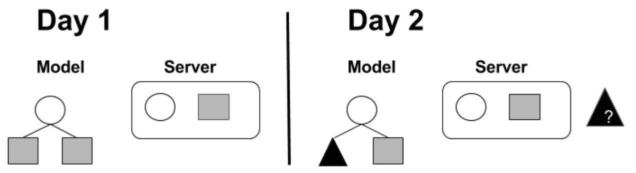
How? The integration test should run both continuously as well as with new releases of models or servers, in order to catch problems well before they reach production. Faster running integration tests with a subset of training data or a simpler model can give faster feedback to developers while still backed by less frequent, long running versions with a setup that more closely mirrors production.

Infra 4: Model quality is validated before attempting to serve it: After a model is trained but before it actually affects real traffic, an automated system needs to inspect it and verify that its quality is sufficient; that system must either bless the model or veto it, terminating its entry to the production environment.

How? It is important to test for both slow degradations in quality over many versions as well as sudden drops in a new version. For the former, setting loose thresholds and comparing against predictions on a validation set can be useful; for the latter, it is useful to compare predictions to the previous version of the model while setting tighter thresholds.

Infra 5: The model allows debugging by observing the step-by-step computation of training or inference on a single example: When someone finds a case where a model is behaving bizarrely, how difficult is it to figure out why? Is there an easy, well documented process for feeding a single example to the model and investigating the computation through each stage of the model (e.g. each internal node of a neural network)?

Figure 2. **Importance of a Model Canary before Serving.** It is possible for models to incorporate new pieces of code that are not live in separate serving binaries, causing havoc at serving time. Using small scale canary processes can help protect against this.



Observing the step-by-step computation through the model on small amounts of data is an especially useful debugging strategy for issues like numerical instability.

How? An internal tool that allows users to enter examples and see how the a specific model version interprets it can be very helpful. The TensorFlow debugger [17] is one example of such a tool.

Infra 6: Models are tested via a canary process before they enter production serving environments: Offline testing, however extensive, cannot by itself guarantee the model will perform well in live production settings, as the real world often contains significant non-stationarity or other issues that limit the utility of historical data. Consequently, there is always some risk when turning on a new model in production.

One recurring problem that canarying can help catch is mismatches between model artifacts and serving infrastructure. 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. For example, as shown in Figure 2, a refactoring in the core learning library might change the low-level implementation of an operation *Op* in the model from *Op0.1* to a more efficient implementation, *Op0.2*. A newly trained model will thus expect to be implemented with *Op0.2*; an older deployed server will not include *Op0.2* and so will refuse to load the model.

How? To mitigate the mismatch issue, one approach is testing that a model successfully loads into production serving binaries and that inference on production input data succeeds. To mitigate the new-model risk more generally, one can turn up new models gradually, running old and new models concurrently, with new models only seeing a small fraction of traffic, gradually increased as the new model is observed to behave sanely.

Infra 7: Models can be quickly and safely rolled back to a previous serving version: A model "roll back" procedure is a key part of incident response to many of the issues that can be detected by the monitoring discussed in Section V. 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. Because rolling back is an emergency procedure, operators should practice doing it

Table IV
BRIEF LISTING OF THE SEVEN MONITORING TESTS

1	Dependency changes result in notification.
2	Data invariants hold for inputs.
3	Training and serving are not skewed.
4	Models are not too stale.
5	Models are numerically stable.
6	Computing performance has not regressed.
7	Prediction quality has not regressed.

normally, when not in emergency conditions.

V. MONITORING TESTS FOR ML

It is crucial to know not just that your ML system worked correctly at launch, but that it continues to work correctly over time. An ML system by definition is making predictions on previously unseen data, and typically also incorporates new data over time into training. The standard approach is to monitor the system, i.e. to have a constantly-updated "dashboard" user interface displaying relevant graphs and statistics, and to automatically alert the engineering team when particular metrics deviate significantly from expectations. For ML systems, it is important to monitor serving systems, training pipelines, and input data. Here we recommend specific metrics to monitor throughout the system. The usual sorts of incident response approaches will apply; one unique to ML is to roll back not the system code but the learned model, hence our test earlier (test Infra 7) to regularly ensure that this process is safe and easy.

Monitor 1: Dependency changes result in notification: ML systems typically consume data from a wide array of other systems to generate useful features. Partial outages, version upgrades, and other changes in the source system can radically change the feature's meaning and thus confuse the model's training or inference, without necessarily producing values that are strange enough to trigger other monitoring.

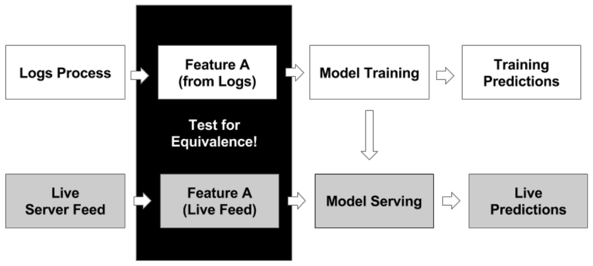
How? Make sure that your team is subscribed to and reads announcement lists for all dependencies, and make sure that the dependent team knows your team is using the data.

Monitor 2: Data invariants hold in training and serving inputs: It can be difficult to effectively monitor the internal behavior of a learned model for correctness, but the input data should be more transparent. Consequently, analyzing and comparing data sets is the first line of defense for detecting problems where the world is changing in ways that can confuse an ML system.

How? Using the schema constructed in test Data 1, measure whether data matches the schema and alert when they diverge significantly. In practice, careful tuning of alerting thresholds is needed to achieve a useful balance between false positive and false negative rates to ensure these alerts remain useful and actionable.

Monitor 3: Training and serving features compute the same values: The codepaths that actually generate input features may differ at training and inference time. Ideally

Figure 3. **Monitoring for Training/Serving Skew.** It is often necessary for the same feature to be computed in different ways in different parts of the system. In such cases, we must carefully test that these different codepaths are in fact logically identical.



the different codepaths should generate the same values, but in practice a common problem is that they do not. This is sometimes called "training/serving skew" and requires careful monitoring to detect and avoid. As one concrete example, imagine adding a new feature to an existing production system. While the value of the feature in the serving system might be computed based on data from live user behavior, the feature will not be present in training data, and so must be backfilled by imputing it from other stored data, likely using an entirely independent codepath. Another example is when the computation at training time is done using code that is highly flexible (for easy experimentation) but inefficient, while at serving time the same computation is heavily optimized for low latency.

How? To measure this, it is crucial to log a sample of actual serving traffic. For systems that use serving input as future training data, adding identifiers to each example at serving time will allow direct comparison; the feature values should be perfectly identical at training and serving time for the same example. Important metrics to monitor here are the number of features that exhibit skew, and the number of examples exhibiting skew for each skewed feature.

Another approach is to compute distribution statistics on the training features and the sampled serving features, and ensure that they match. Typical statistics include the minimum, maximum, or average, values, the fraction of missing values, etc. Again, thresholds for alerting on these metrics must be carefully tuned to ensure a low enough false positive rate for actionable response.

Monitor 4: Models are not too stale: In test Model 4 we discussed testing the effect that an old ("stale") model has on prediction quality. Here, we recommend monitoring how old the system in production is, using the prior measurement as a guide for determining what age is problematic enough to raise an alert.

Surprisingly, infrequently updated models also incur a maintenance cost. Imagine a model that is manually retrained once or twice a year by a given engineer. If that engineer leaves the team, this process may be difficult to replicate – even carefully written instructions may become stale or incorrect over this kind of time horizon.

How? For models that re-train regularly (e.g. weekly or more often), the most obvious metric is the age of the model in production. It is also important to measure the age of the model at each stage of the training pipeline, to quickly determine where a stall has occurred and react appropriately.

Even for models that re-train more infrequently, there is often a dependence on data aggregation or other such processes to produce features, which can themselves grow stale. For example, consider using a feature based on the most popular n items (movies, apps, cars, etc). The process that computes the top-n table must be re-run frequently, and it is crucial to monitor the age of this table, so that if the process stops running, alerts will fire.

Monitor 5: The model is numerically stable: Invalid or implausible numeric values can potentially crop up during model training without triggering explicit errors, and knowing that they have occurred can speed diagnosis of the problem.

How? Explicitly monitor the initial occurrence of any NaNs or infinities. Set plausible bounds for weights and the fraction of ReLU units in a layer returning zero values, and trigger alerts during training if these exceed appropriate thresholds.

Monitor 6: The model has not experienced a 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. Deep neural networks can be slow to train and run inference on, wide linear models with feature crosses can use a lot of memory; any ML model may take days to train; and so forth. Swiftly reacting to changes in this performance due to changes in data, features, modeling, or underlying compute library or infrastructure is crucial to maintaining a performant system.

How? While measuring computational performance is a standard part of any monitoring, it is useful to slice performance metrics not just by the versions and components of code, but also by data and model versions. Degradations in computational performance may occur with dramatic changes (for which comparison to performance of prior versions or time slices can be helpful for detection) or in slow leaks (for which a pre-set alerting threshold can be helpful for detection)

Monitor 7: The model has not experienced a regression in prediction quality on served data: Validation data will always be older than real serving input data, so measuring a model's quality on that validation data before pushing it to serving is only an estimate of quality metrics on actual live serving inputs. However, it is not always possible to know the correct labels even shortly after serving time, making quality measurement difficult.

How? Here are some options to make sure that there is no degradation in served prediction quality due to changes in data, differing codepaths, etc.

- Measure statistical bias in predictions, i.e. the average of predictions in a particular slice of data. Generally speaking, models should have zero bias, in aggregate and on slices (e.g. 90% of predictions of probability 0.9 should in fact be positive). Knowing that a model is unbiased is not enough to know it is any good, but knowing there is bias can be a useful canary to detect problems.
- In some tasks, the label actually is available immediately or soon after the prediction is made (e.g. will a user click on an ad). In this case, we can judge the quality of predictions in almost real-time and identify problems quickly.
- Finally, it can be useful to periodically add new training data by having human raters manually annotate labels for logged serving inputs. Some of this data can be held out to validate the served predictions.

However the measure can be done, thresholds must be set as to acceptable quality (e.g. based on bounds of quality at the launch of the initial system), and then a responder should be notified immediately if quality drifts outside that threshold. As with computational performance, it is crucial to monitor both dramatic and slow-leak regressions in prediction quality.

VI. Incentivizing culture change

Because technical debt is difficult to quantify, it can be difficult to prioritize paydown or measure improvements. To address this, our rubric provides a quantified *ML Test Score* which can be measured and improved over time. This provides a vector for incentivizing ML system developers to achieve strong levels of reliability by providing a clear indicator of readiness and clear guidelines for how to improve. This strategy was inspired by the Test Certified program at Google, which provided a scored ladder for overall test robustness, and which had strong success in incentivizing teams to adopt best practices.

A. Computing an ML Test Score

The final test score is computed as follows:

- For each test, half a point is awarded for executing the test manually, with the results documented and distributed.
- A full point is awarded if there is a system in place to run that test automatically on a repeated basis.
- Sum the score for each of the 4 sections individually.
- The final ML Test Score is computed by taking the *minimum* of the scores aggregated for each of the 4 sections.

We choose the minimum because we believe all four sections are important, and so a system must consider all in order to raise the score. One downside of this approach is that it reduces the extent to which an individual's efforts are reflected in higher system scores and ranks; it remains to be seen how this will affect the adoption of our system.

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

Table V
Interpreting an ML Test Score. THIS SCORE IS COMPUTED BY TAKING THE *minimum* SCORE FROM EACH OF THE FOUR TEST AREAS. NOTE THAT DIFFERENT SYSTEMS AT DIFFERENT POINTS IN THEIR DEVELOPMENT MAY REASONABLY AIM TO BE AT DIFFERENT POINTS ALONG THIS SCALE.

All tests are worth the same number of points. This is intentional, as we believe the relative importance of tests to teams will vary depending on their specific priorities. This means that choosing any test to implement will raise the score, and we feel that is appropriate, as they are each valuable and often working on one will make it easier to work on another.

To interpret the score, see Table V. These interpretations were calibrated against a number of internal ML systems, and overall have been reflective of other qualitative perceptions of those systems.

VII. APPLYING THE RUBRIC TO REAL SYSTEMS

We developed the ML Test Certified program to help engineers doing ML work at Google. Some of our work has involved meeting with teams doing ML and evaluating their performance in a structured interview based on the rubric detailed above. We met with 36 teams from across Google working in a diverse array of product areas; their scores on the rubric are presented in Figure 4. These interviews have offered some unexpected insights.

A. *The importance of checklists*

Checklists are helpful even for expert teams [18]. For example, one team we worked with discovered a thousand-line code file, completely untested, that created their input features. Code of that size, even if it contains only simple and straightforward logic, will likely have bugs, against which simple unit tests can provide an effective hedge. Another example we found was a team who realized when we asked that they had no evaluation or monitoring to discover if their global service was serving poor predictions localized to a single country. They also relied heavily on informal evaluation of performance based on the team's own usage of the product, which does not protect users very different from the team members. Similarly, the interviews were useful simply as a way of advertising the existing tools – some teams had not even heard of the Facets tools or of our unit testing framework mentioned in Infra 2.

As another example, when we asked one team about ML inclusiveness, they confidently answered that they had given the matter some thought and concluded that there was no way for their system to be biased since they were only dealing with speech waveforms ("we just get vectors of numbers"). When we asked if they had done any work to ensure their system performed well for African American Vernacular English or had taken steps to ensure diversity in the population of human raters they hired for scoring, they paused at length and then agreed that this question opened up new possibilities for debiasing which they had not considered and would address.

Finally, the context of our interview provided additional motivation for getting around to implementing tests - one team was motivated to implement feature code tests because of the clear danger of training/serving skew, while others were spurred to automate previously manual processes to make them more frequent and testable.

B. *Dependency issues*

Data dependencies can lead to outsourcing responsibility for fully understanding it. Multiple teams initially suggested that since their features were produced by an upstream, much larger service, any problems in their data would be discovered by the other team. While this can certainly be some protection, it may still be that the smaller team has different requirements for the data that would not be caught by the larger team's validation.

In the other direction, multiple teams initially suggested that their system did not require independent monitoring, as their serving was done via a larger system whose reliability engineers would notice any problems downstream. Again, this can be some protection, but it's also quite possible that the smaller system's errors may be masked in the noise of the larger system. In addition, it's crucial in that regime that the larger system know how to find the appropriate contact person from the smaller one.

For the data tests, several teams indicated a key distinction between features that represent new combinations of existing data sources, and features based on new data sources. The latter requires significantly more time and introduces more risk. Depending on a new data source can mean time spent negotiating with the owning team to ensure the data is properly treated. Or if the data come from newly logged information, the existing training data must be backfilled, or thrown away to wait for new logs including the data.

C. *The importance of frameworks*

Integration testing (Infra 3) stood out as a test with much lower adoption than most. When implemented, it often

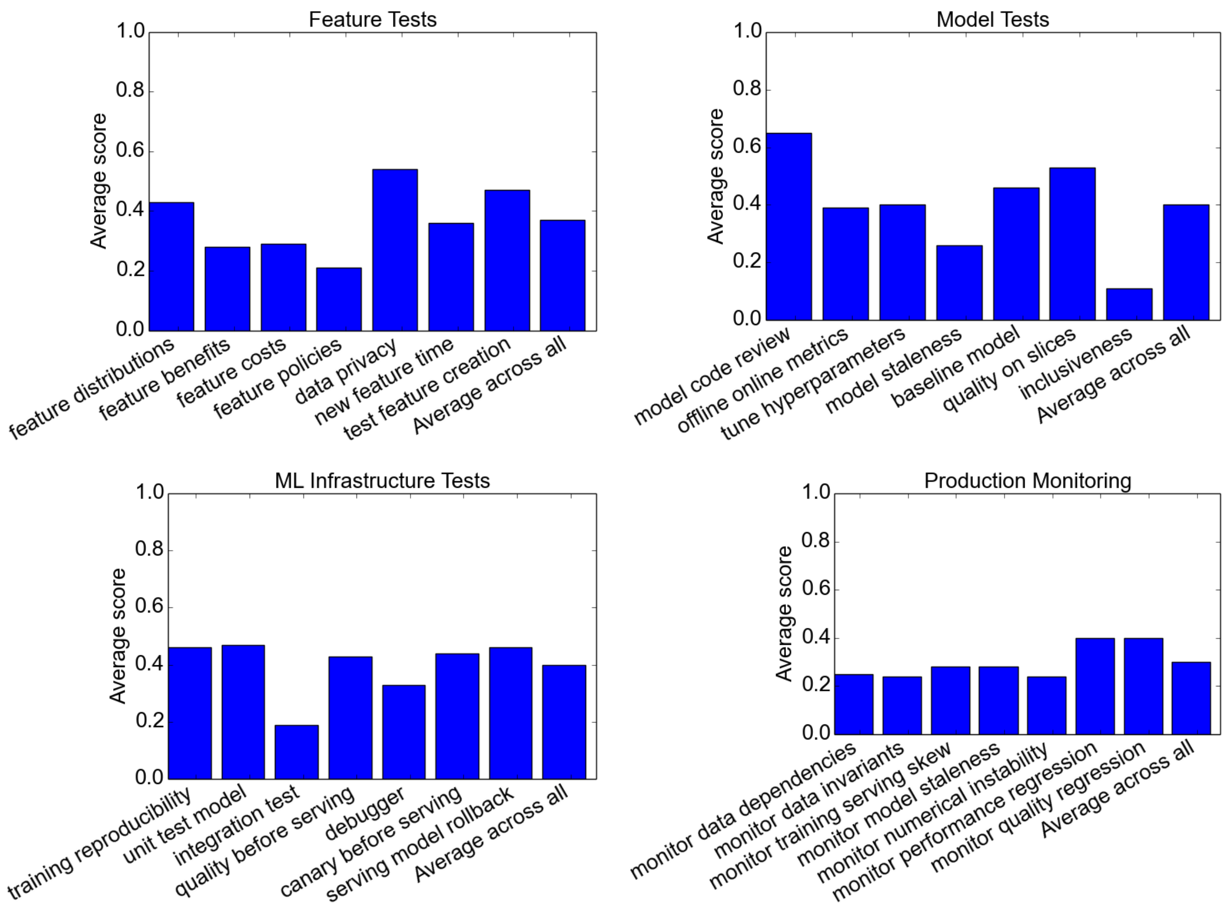


Figure 4. **Average scores for interviewed teams.** These graphs display the average score for each test across the 36 systems we examined.

included serving systems but not training. This is in part because training is often developed as an ad hoc set of scripts and manual processes. A training pipeline platform like the TFX system[19] can be beneficial here as it then allows building a generic integration test.

Model canarying (Infra 6) was frequently implemented by many teams, and cited as a key part of their testing plan. But this masks two interesting issues. First, canarying can indeed catch many issues like unservable models, numeric instability, and so forth. However, it typically occurs long after the engineering decisions that led to the issue, so it would be much preferable to catch issues earlier in unit or integration tests. Second, the teams that implemented canarying usually did so because their existing release framework made it easy – and one team lacking such a framework reported the one time they did canary it was so painful they'd never do it again.

Perhaps the most important and least implemented test is the one for training/serving skew (Monitor 3). This sort of error is responsible for production issues across a wide swath of teams, and yet it is one of the least frequently implemented tests. In part this is because it is difficult, but again, building this into a framework like TFX allows many teams to benefit from a single investment.

To test TFX, we evaluated a hypothetical system that used TFX along with its standard recommendations for introductory data analysis and so forth. We found that this hypothetical system already scored as "reasonably tested" according to our criterion. TFX is quite new, however, and we haven't yet measured real world TFX systems.

D. *Assessing the assessment*

We also conducted some meta-level assessment, asking teams what was useful or non-useful about this rubric.

One interesting theme was that teams using purely image or audio data did not feel many of the Data tests were applicable. However, methods like manual inspection of raw data and LIME-style importance analysis [20] remain important tools in such settings. For example, such inspection can reveal skew in distributions or unrealistically consistent background effects correlated with the training target.

Supervised ML requires labeled data, but a number of groups are working in domains where labels are either not present or extremely expensive to acquire. One group had an extremely large data set that was so diverse that using human raters to generate training labels proved infeasible. So they built a simple heuristic system and then used that to train an ML system ("The ML experts told us that training a model like this was crazy and would never work but they were wrong!"). Human raters consistently rate the heuristic system as good but the ML system trained from it as much better – however, this exposes a need for a level of testing of the base heuristic system that is not covered in our rubric. Expensive labels also mean that quality evaluation of a learned model is difficult, which impacts the ability of teams to implement several tests like Model 4 and Infra 4.

Acknowledgment

We are very grateful to Keith Arner, Gary Holt, Josh Lovejoy, Fernando Pereira, Todd Phillips, Tal Shaked, Todd Underwood, Martin Wicke, Cory Williams, and Martin Zinkevich for many helpful discussions and comments on drafts of this paper.

References

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high interest credit card of technical debt," in *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[2] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica, "Ad click prediction: A view from the trenches," in *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '13. New York, NY, USA: ACM, 2013, pp. 1222–1230. [Online]. Available: http://doi.acm.org/10.1145/2487575.2488200

[3] T. Chandra, E. Ie, K. Goldman, T. L. Llinares, J. McFadden, F. Pereira, J. Redstone, T. Shaked, and Y. Singer, "Sibyl: a system for large scale machine learning," vol. 28, Jul. 2010.

[4] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and software technology*, vol. 56, no. 10, pp. 1219–1232, 2014.

[5] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications." in *SEKE*. Citeseer, 2007, p. 167.

[6] M. Zinkevich, "Rules of machine learning," Invited talk at the NIPS Reliable Machine Learning Workshop, 1996. [Online]. Available: http://martin.zinkevich.org/rules_of_ml/rules_of_ml.pdf

[7] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, "The ML test score: A rubric for ML production readiness and technical debt reduction," in *Proceedings of IEEE Big Data 2017*, 2017.

[8] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, "The daikon system for dynamic detection of likely invariants," *Science of Computer Programming*, vol. 69, no. 1, pp. 35–45, 2007.

[9] C.-W. Hsu, C.-C. Chang, C.-J. Lin *et al.*, "A practical guide to support vector classification," 2003.

[10] J. Snoek, H. Larochelle, and R. P. Adams, "Practical bayesian optimization of machine learning algorithms," in *Advances in neural information processing systems*, 2012.

[11] T. Desautels, A. Krause, and J. Burdick, "Parallelizing exploration-exploitation tradeoffs in gaussian process bandit optimization," *Journal of Machine Learning Research (JMLR)*, vol. 15, p. 40534103, December 2014.

[12] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *KDD 2017*, 2017.

[13] "Google cloud machine learning: now open to all with new professional services and education programs," https://goo.gl/ULh7ZW, 2017, accessed: 2017-02-08.

[14] T. Bolukbasi, K.-W. Chang, J. Y. Zou, V. Saligrama, and A. T. Kalai, "Man is to computer programmer as woman is to homemaker? debiasing word embeddings," in *Advances in Neural Information Processing Systems 29*, D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, Eds., 2016.

[15] M. Hardt, E. Price, N. Srebro *et al.*, "Equality of opportunity in supervised learning," in *Advances in Neural Information Processing Systems*, 2016, pp. 3315–3323.

[16] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, M. aurelio Ranzato, A. Senior, P. Tucker, K. Yang, Q. V. Le, and A. Y. Ng, "Large scale distributed deep networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1223–1231. [Online]. Available: http://papers.nips.cc/paper/4687-large-scale-distributed-deep-networks.pdf

[17] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "Tensorflow debugger: Debugging dataflow graphs for machine learning," in *Proceedings of the Reliable Machine Learning in the Wild - NIPS 2016 Workshop*, 2016.

[18] A. Gawande, *Checklist Manifesto, The*. Henry Holt and Company, 2009.

[19] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich, "Tfx: A tensorflow-based production-scale machine learning platform," in *KDD 2017*, 2017.

[20] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should I trust you?": Explaining the predictions of any classifier," *CoRR*, vol. abs/1602.04938, 2016. [Online]. Available: http://arxiv.org/abs/1602.04938