DATA SCIENCE

Effective testing for machine learning systems.

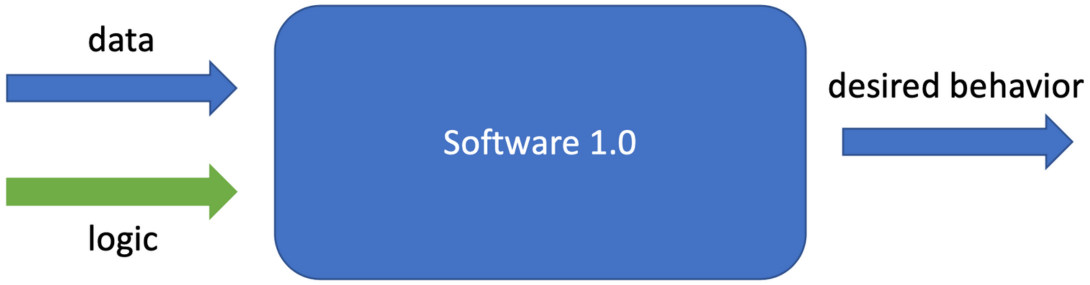
JEREMY JORDAN
19 AUG 2020 • 9 MIN READ

Working as a core maintainer for PyTorch Lightning, I've grown a strong appreciation for the value of tests in software development. As I've been spinning up a new project at work, I've been spending a fair amount of time thinking about how we should test machine learning systems. A couple weeks ago, one of my coworkers sent me a fascinating paper on the topic which inspired me to dig in, collect my thoughts, and write this blog post.

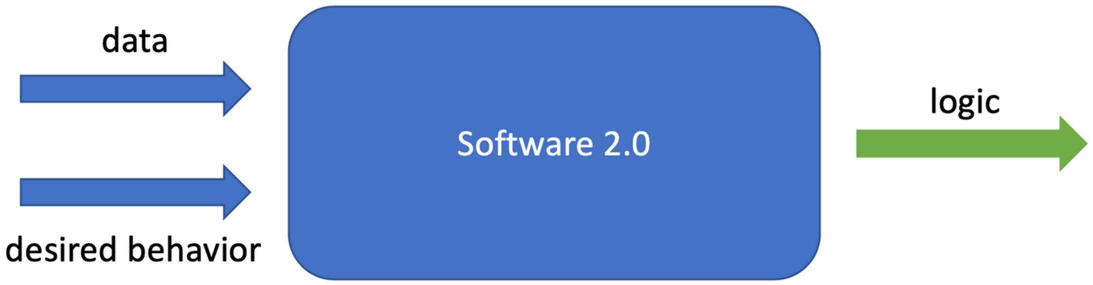
In this blog post, we'll cover what testing looks like for traditional software development, why testing machine learning systems can be different, and discuss some strategies for writing effective tests for machine learning systems. We'll also clarify the distinction between the closely related roles of evaluation and testing as part of the model development process. By the end of this blog post, I hope you're convinced of both the extra work required to effectively test machine learning systems and the value of doing such work.

learning systems?

In traditional software systems, humans write the logic which interacts with data to produce a desired behavior. Our software tests help ensure that this **written logic** aligns with the actual expected behavior.



However, in machine learning systems, humans provide desired behavior as examples during training and the model optimization process produces the logic of the system. How do we ensure this **learned logic** is going to consistently produce our desired behavior?



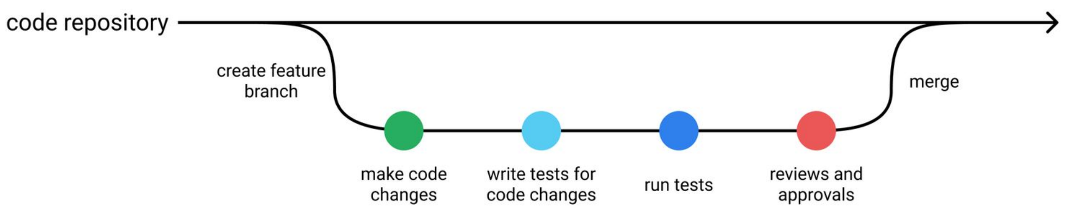
Let's start by looking at the best practices for testing traditional software systems and developing high-quality software.

A typical software testing suite will include:

- **regression tests** replicate bugs that we've previously encountered and fixed,
- **integration tests** which are typically longer-running tests that observe higher-level behaviors that leverage multiple components in the codebase,

and follow conventions such as:

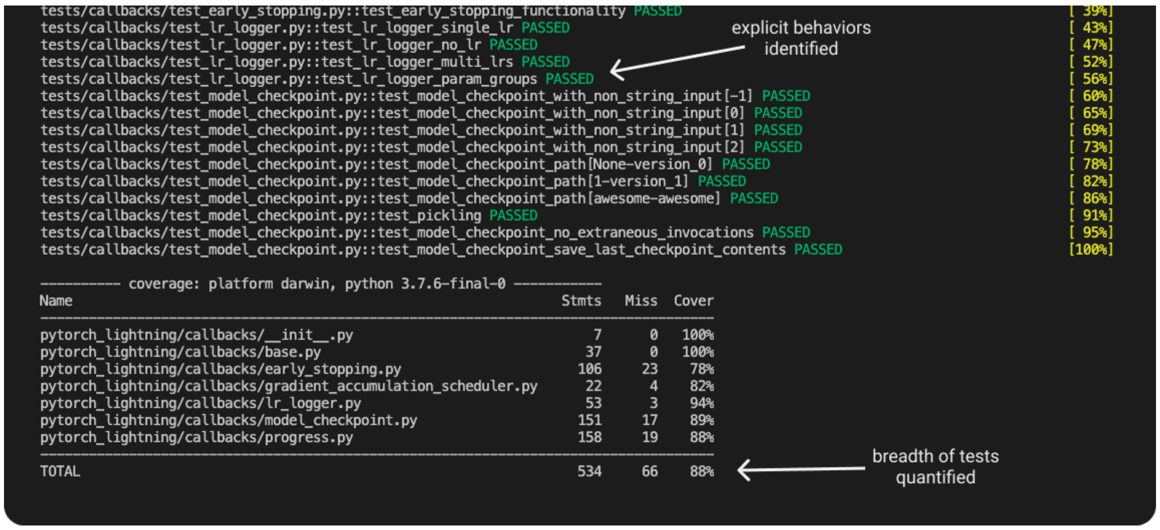
- don't merge code unless all tests are passing,
- always write tests for newly introduced logic when contributing code,
- when contributing a bug fix, be sure to write a test to capture the bug and prevent future regressions.



A typical workflow for software development.

When we run our testing suite against the new code, we'll get a report of the specific behaviors that we've written tests around and verify that our code changes don't affect the expected behavior of the system. If a test fails, we'll know which specific behavior is no longer aligned with our expected output. We can also look at this testing report to get an understanding of how extensive our tests are by looking at metrics such as **code coverage**.

```
tests/callbacks/test_callbacks.py::test_trainer_callback_system PASSED                  [  4%]
```



```
tests/callbacks/test_early_stopping.py::test_early_stopping_functionality PASSED                          [ 39%]
tests/callbacks/test_lr_logger.py::test_lr_logger_single_lr PASSED                                          [ 43%]
tests/callbacks/test_lr_logger.py::test_lr_logger_no_lr PASSED                                              [ 47%]
tests/callbacks/test_lr_logger.py::test_lr_logger_multi_lrs PASSED                                          [ 52%]
tests/callbacks/test_lr_logger.py::test_lr_logger_param_groups PASSED                                       [ 56%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_with_non_string_input[-1] PASSED            [ 60%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_with_non_string_input[0] PASSED             [ 65%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_with_non_string_input[1] PASSED             [ 69%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_with_non_string_input[2] PASSED             [ 73%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_path[None-version_0] PASSED                  [ 78%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_path[1-version_1] PASSED                     [ 82%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_path[awesome-awesome] PASSED                 [ 86%]
tests/callbacks/test_model_checkpoint.py::test_pickling PASSED                                              [ 91%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_no_extraneous_invocations PASSED            [ 95%]
tests/callbacks/test_model_checkpoint.py::test_model_checkpoint_save_last_checkpoint_contents PASSED        [100%]

---------- coverage: platform darwin, python 3.7.6-final-0 -----------
Name                                                      Stmts   Miss  Cover
-----------------------------------------------------------------------------
pytorch_lightning/callbacks/__init__.py                       7      0   100%
pytorch_lightning/callbacks/base.py                          37      0   100%
pytorch_lightning/callbacks/early_stopping.py               106     23    78%
pytorch_lightning/callbacks/gradient_accumulation_scheduler.py 22    4    82%
pytorch_lightning/callbacks/lr_logger.py                     53      3    94%
pytorch_lightning/callbacks/model_checkpoint.py             151     17    89%
pytorch_lightning/callbacks/progress.py                     158     19    88%
-----------------------------------------------------------------------------
TOTAL                                                       534     66    88%
```

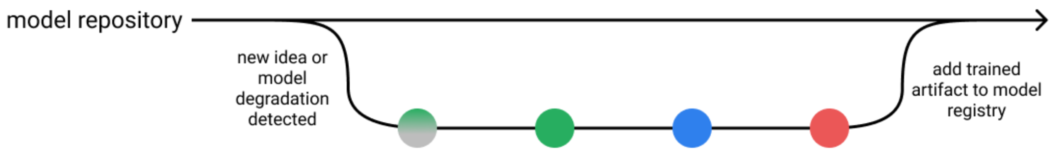
An example output from a traditional software testing suite.

Let's contrast this with a typical workflow for developing machine learning systems. After training a new model, we'll typically produce an evaluation report including:

- performance of an established metric on a validation dataset,
- plots such as precision-recall curves,
- operational statistics such as inference speed,
- examples where the model was most confidently incorrect,

and follow conventions such as:

- save all of the hyper-parameters used to train the model,
- only promote models which offer an improvement over the existing model (or baseline) when evaluated on the same dataset.


model repository
new idea or model degradation detected
add trained artifact to model registry

When reviewing a new machine learning model, we'll inspect metrics and plots which summarize model performance over a validation dataset. We're able to compare performance between multiple models and make relative judgements, but we're not immediately able to characterize specific model behaviors. For example, figuring out *where* the model is failing usually requires additional investigative work; one common practice here is to look through a list of the top most egregious model errors on the validation dataset and manually categorize these failure modes.

Assuming we write behavioral tests for our models (discussed below), there's also the question of whether or not we have enough tests! While traditional software tests have metrics such as the lines of code covered when running tests, this becomes harder to quantify when you shift your application logic from lines of code to parameters of a machine learning model. Do we want to quantify our test coverage with respect to the input data distribution? Or perhaps the possible activations inside the model?

Odena et al. introduce one possible metric for coverage where we track the model logits for all of the test examples and quantify the area covered by radial neighborhoods around these activation vectors. However, my perception is that as an industry we don't have a well-established convention here. In fact, it feels like that testing for machine learning systems is in such early days that this question of test coverage isn't really being asked by many people.

What's the difference between model testing and model evaluation?

sufficient. Without a granular report of specific behaviors, we won't be able to immediately understand the nuance of how behavior may change if we switch over to the new model. Additionally, we won't be able to track (and prevent) behavioral regressions for specific failure modes that had been previously addressed.

This can be especially dangerous for machine learning systems since often times failures happen silently. For example, you might improve the overall evaluation metric but introduce a regression on a critical subset of data. Or you could unknowingly add a gender bias to the model through the inclusion of a new dataset during training. We need more nuanced reports of model behavior to identify such cases, which is exactly where model testing can help.

For machine learning systems, we should be running model evaluation and model tests in parallel.

- **Model evaluation** covers metrics and plots which summarize performance on a validation or test dataset.
- **Model testing** involves explicit checks for behaviors that we expect our model to follow.

Both of these perspectives are instrumental in building high-quality models.

In practice, most people are doing a combination of the two where evaluation metrics are calculated automatically and some level of model "testing" is done manually through error analysis (i.e. classifying failure modes). Developing model tests for machine learning systems can offer a systematic approach towards error analysis.

In my opinion, there's two general classes of model tests that we'll want to write.

- **Pre-train tests** allow us to identify some bugs early on and short-circuit a training job.
- **Post-train tests** use the trained model artifact to inspect behaviors for a variety of important scenarios that we define.

Pre-train tests

There's some tests that we can run without needing trained parameters. These tests include:

- check the shape of your model output and ensure it aligns with the labels in your dataset
- check the output ranges and ensure it aligns with our expectations (eg. the output of a classification model should be a distribution with class probabilities that sum to 1)
- make sure a single gradient step on a batch of data yields a decrease in your loss
- make assertions about your datasets
- check for label leakage between your training and validation datasets

The main goal here is to identify some errors early so we can avoid a wasted training job.

Post-train tests

However, in order for us to be able to understand model behaviors we'll a behavioral report of model performance.

> **Paper highlight**: Beyond Accuracy: Behavioral Testing of NLP Models with CheckList

The authors of the above paper present three different types of model tests that we can use to understand behavioral attributes.

Invariance Tests

Invariance tests allow us to describe a set of perturbations we should be able to make to the input *without* affecting the model's output. We can use these perturbations to produce pairs of input examples (original and perturbed) and **check for consistency** in the model predictions. This is closely related to the concept of data augmentation, where we apply perturbations to inputs during training and preserve the original label.

For example, imagine running a sentiment analysis model on the following two sentences:

- Mark was a great instructor.
- Samantha was a great instructor.

We would expect that simply changing the name of the subject doesn't affect the model predictions.

Directional Expectation Tests

Directional expectation tests, on the other hand, allow us to define a set of perturbations to the input which *should* have a *predictable* effect on assert:

- Increasing the number of bathrooms (holding all other features constant) should not cause a drop in price.
- Lowering the square footage of the house (holding all other features constant) should not cause an increase in price.

Let's consider a scenario where a model fails the second test - taking a random row from our validation dataset and decreasing the feature `house_sq_ft` yields a higher predicted price than the original label. This is surprising as it doesn't match our intuition, so we decide to look further into it. We realize that, without having a feature for the house's neighborhood/location, our model has learned that smaller units tend to be more expensive; this is due to the fact that smaller units from our dataset are more prevalent in cities where prices are generally higher. In this case, the *selection* of our dataset has influenced the model's logic in unintended ways - this isn't something we would have been able to identify simply by examining performance on a validation dataset.

Minimum Functionality Tests (aka data unit tests)

Just as software unit tests aim to isolate and test atomic components in your codebase, data unit tests allow us to quantify model performance for specific cases found in your data.

This allows you to identify critical scenarios where prediction errors lead to high consequences. You may also decide to write data unit tests for failure modes that you uncover during error analysis; this allows you to "automate" searching for such errors in future models.

Snorkel also has introduced a very similar approach through their example, you might write a slicing function to identify sentences less than 5 words to evaluate how the model performs on short pieces of text.

Organizing tests

In traditional software tests, we typically organize our tests to mirror the structure of the code repository. However, this approach doesn't translate well to machine learning models since our logic is structured by the parameters of the model.

The authors of the CheckList paper linked above recommend structuring your tests around the "skills" we expect the model to acquire while learning to perform a given task.

For example, a sentiment analysis model might be expected to gain some understanding of:

- vocabulary and parts of speech,
- robustness to noise,
- identifying named entities,
- temporal relationships,
- and negation of words.

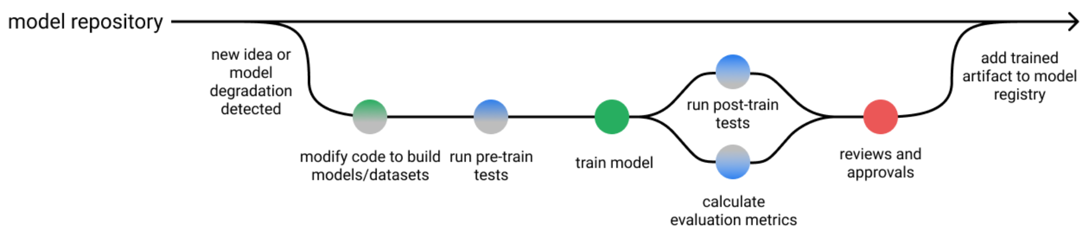
For an image recognition model, we might expect the model to learn concepts such as:

- object rotation,
- partial occlusion,
- perspective shift,

- and camera artifacts (ISO noise, motion blur).

Model development pipeline

Putting this all together, we can revise our diagram of the model development process to include pre-train and post-train tests. These tests outputs can be displayed alongside model evaluation reports for review during the last step in the pipeline. Depending on the nature of your model training, you may choose to automatically approve models provided that they meet some specified criteria.


model repository
new idea or model degradation detected
modify code to build models/datasets
run pre-train tests
train model
run post-train tests
calculate evaluation metrics
reviews and approvals
add trained artifact to model registry

A proposed workflow for developing high-quality models.

Conclusion

Machine learning systems are trickier to test due to the fact that we're not explicitly writing the logic of the system. However, automated testing is still an important tool for the development of high-quality software systems. These tests can provide us with a behavioral report of trained models, which can serve as a systematic approach towards error analysis.

Throughout this blog post, I've presented "traditional software development" and "machine learning model development" as two separate concepts. This simplification made it easier to discuss the unique challenges associated with testing machine learning systems;

development" in order to process data inputs, create feature representations, perform data augmentation, orchestrate model training, expose interfaces to external systems, and much more. Thus, effective testing for machine learning systems requires **both** a traditional software testing suite (for model development infrastructure) and a model testing suite (for trained models).

If you have experience testing machine learning systems, please reach out and share what you've learned!

Thank you, Xinxin Wu, for sending me the paper which inspired me to write this post! Additionally, I'd like to thank John Huffman, Josh Tobin, and Andrew Knight for reading earlier drafts of this post and providing helpful feedback.

Further reading

Papers

- Beyond Accuracy: Behavioral Testing of NLP Models with CheckList
- TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing

Blog posts

- How to Test Machine Learning Code and Systems by Eugene Yan
- Snorkel Intro Tutorial: *Data Slicing*
- How to Trust Your Deep Learning Code by Tilman Krokotsch

- MLOps Chat: How Should We Test ML Models? with Data Scientist Jeremy Jordan
- Unit Testing for Data Scientists - Hanna Torrence
- Trey Causey: Testing for Data Scientists
- Bay Area NLP Meetup: Beyond Accuracy Behavioral Testing of NLP Models with CheckList

Code

- CheckList
- Great Expectations
- Chex (testing library for Jax)

An introduction to Kubernetes.

2 years ago • 7 comments

This blog post will provide an introduction to Kubernetes so that you …

Neural networks: activation functions.

4 years ago • 3 comments

Activation functions are used to determine the firing of neurons in a neural …

Common architectures in convolutional …

4 years ago • 4 comments

In this post, I'll discuss commonly used architectures for …

Jeremy Jordan – Effective testing for machine learning syste Subscribe

Favorite 10 Tweet Share Sort by Best

Join the discussion…

LOG IN WITH

OR SIGN UP WITH DISQUS ?

Name

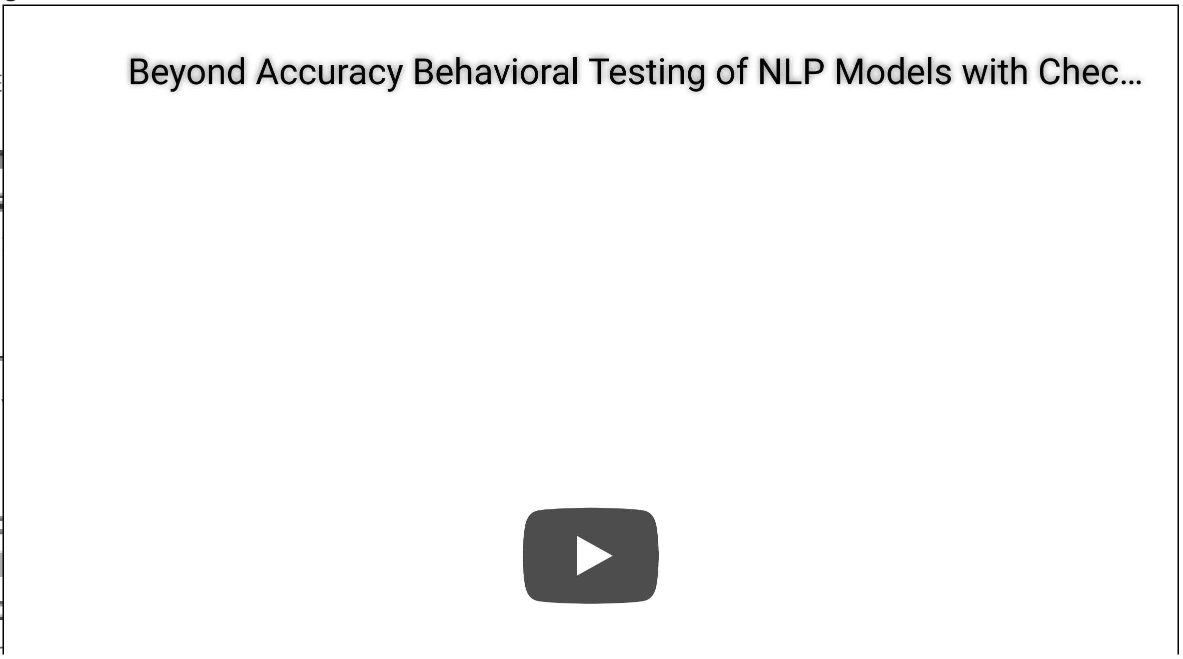
Markus Neifer • a year ago
Excellent overview of the topic. Thanks a lot for sharing.
1 | 1 • Reply • Share ›

Darío López Padial • 2 months ago
Fantastic article, Jeremy!

Do you know more open source projects applying this kind of testing?
| • Reply • Share ›

Othman Alikhan • a year ago
Thanks for sharing, very interesting read. I'm glad more and more attention is being given to testing and quality assurance =)
| • Reply • Share ›

Robert Munro • a year ago
The authors of "Beyond Accuracy..." presented the paper at my meetup last week and gave a live Q&A afterwards:



MORE

A sin
ML s
2 Jan

An ir
26 No

Builc
prod
is a p
21 Sep

DATA SCIENCE

An introduction to Kubernetes.

This blog post will provide an introduction to Kubernetes so that you can understand the motivation behind the tool, what it is, and how you can use it. In a follow-up post, I'll discuss how we can leverage Kubernetes to power data science workloads

JEREMY JORDAN
26 NOV 2019 • 15 MIN READ