FSDL Lecture 10: Testing and Explaining Machine Learning Systems

Video and Slides by Josh Tobin - posted on the FSDL Course Website
Notes were taken by James Le and Vishnu Rachakonda

1 - What's Wrong With Black-Box Predictions?

What does it mean when we have a good test set performance?

*If the test data and production data come from the **same distribution**, then **in expectation**, the performance **of your model** on **your evaluation metrics** will be the same.*

Let's unpack the bolded assumptions:

- **In the real world, the production distribution does not always match the offline distribution**. You could have data drift, data shift, or even malicious users trying to attack your model.
- **Expected performance does not tell the whole story**. For instance, if you are working on long-tail data distribution, then the sample of data that you use to evaluate the model offline might not tell you much about the tail of that distribution - meaning that your test set score can be misleading. On top of that, if you evaluate your model with a single metric across your entire dataset, that does not mean your model is actually performing well against all the slices of data that might be important.
- **The performance of your model is not equal to the performance of your machine learning system**. There are other things (that can go wrong with the ML system) that do not have anything to do with the model.
- Finally, the test set performance only tells you about the metrics that you are evaluating**. In the real world, you are probably not optimizing the exact metrics you care about deep down.**

How bad is this problem? This is a quote from a former ML engineer at an autonomous vehicle company: "*I think the single biggest thing holding back the autonomous vehicle industry today is that, even if we had a car that worked, no one would know, because no one is confident that they know how to evaluate it properly.*" We believe that there is a similar sentiment to lesser degrees in other fields of machine learning, where **the evaluation is the biggest bottleneck**.

The goal of this lecture is to introduce concepts and methods to help you, your team, and your users:

1. Understand at a deeper level how well your model is performing.
2. Become more confident in your model's ability to perform well in production.

3. Understand the model's **performance envelope** (where you should expect it to perform well and where not).

2 - Software Testing

Types of Tests

There are three basic types of software tests:

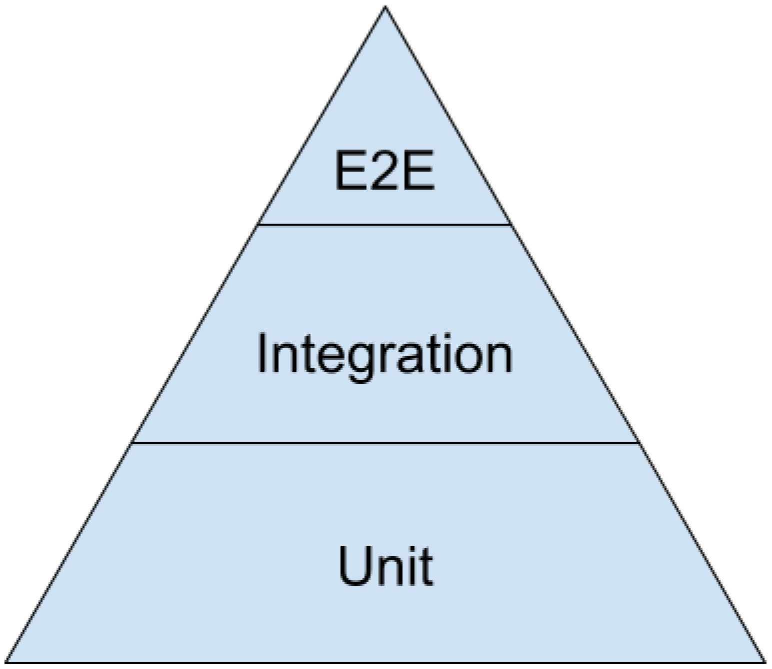
1. **Unit tests** that test the functionality of a single piece of code (an assertion on a single function or a single class) in isolation.
2. **Integration tests** that test how two or more units perform when used together (e.g., test if a model works well with a pre-processing function).
3. **End-to-end tests** that test how the entire software system performs when all units are put together (e.g., test on realistic inputs from a real user).

Testing is a broad field, so you will likely encounter various other kinds of tests as well.

Best Practices

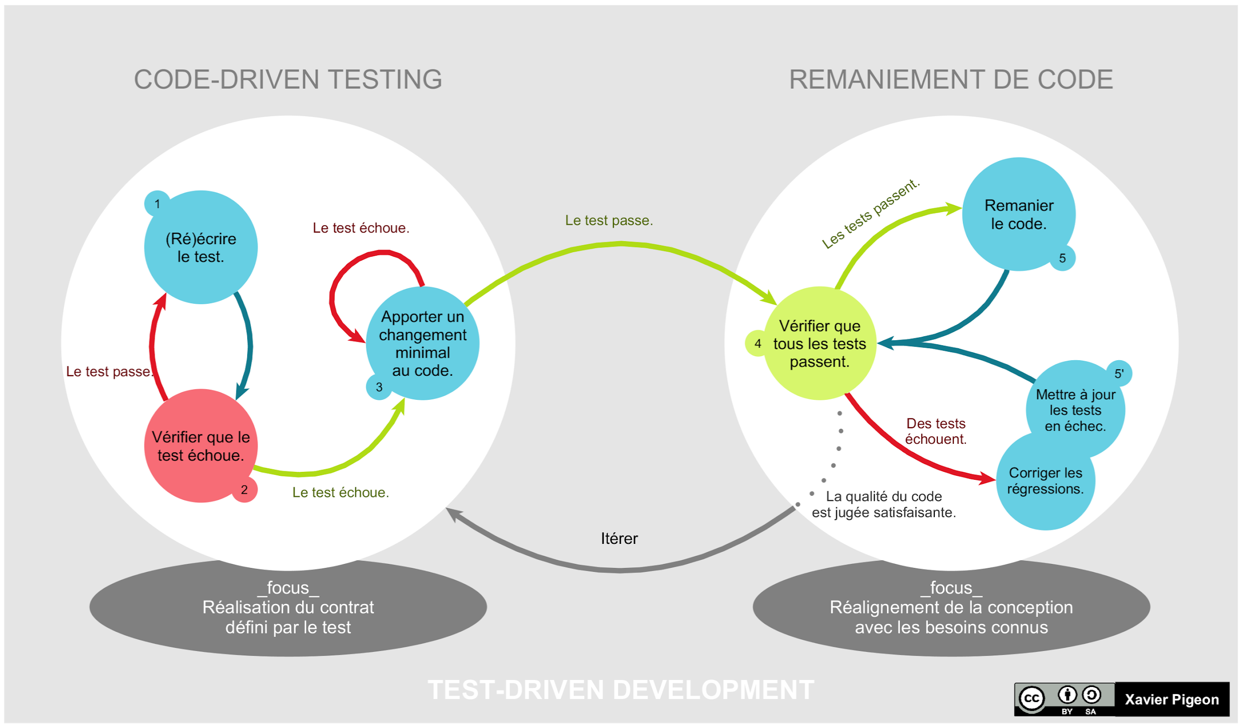
Here are a couple of "uncontroversial" testing best practices:

- **Automate your tests:** You have tests that run by themselves (typically via a CI/CD system) without a user committing an action. There should be no ambiguity on whether your system performs up to standard on the tests that are being run.
- **Make sure your tests are reliable, run fast, and go through the same code review process as the rest of your code:** The number of tests grows in proportion to the size of your codebase. If your tests are unreliable, then people will start ignoring them. If your tests are slow, then you won't want to run them frequently during development. If your tests do not undergo the code review process, they will have bugs, and it's better not to have them at all.
- **Enforce that tests must pass before merging into the main branch:** This is a good norm for teams with more than one person. This is a forcing function to make sure that everyone is committed to writing good tests and can also be helpful for regulatory concerns.
- **When you find new production bugs, convert them to tests:** This ensures that someone does not accidentally reintroduce those bugs in the future.
- **Follow the testing pyramid:** Introduced by Google, it says that you should write a lot more unit tests than integration tests and a lot more integration tests than end-to-end tests. Compared to end-to-end tests, unit tests are faster, more reliable, and better at isolating failures. The rule of thumb that Google recommends (as a rough split) is 70% unit tests, 20% integration tests, and 10% end-to-end tests.



Next up, let’s discuss a few “controversial” testing best practices:

- **Solitary tests:** The distinction between a solitary test and a sociable test is that - solitary testing does not rely on real data from other units, while sociable testing makes the implicit assumption that other modules are working.
- **Test coverage:** You get a test coverage score for your codebase, which tells you what percentage of lines of code in your codebase is called by at least one test. Test coverage gives you a single metric that quantifies the quality of your testing suite. However, test coverage does not measure the right things (in particular, test quality).
- **Test-driven development:** In principle, you want to create your tests before you write your code. These tests serve as the specification of how the code functions. There are not many people who religiously stick to this methodology of development, but TDD is a valuable tool nonetheless.

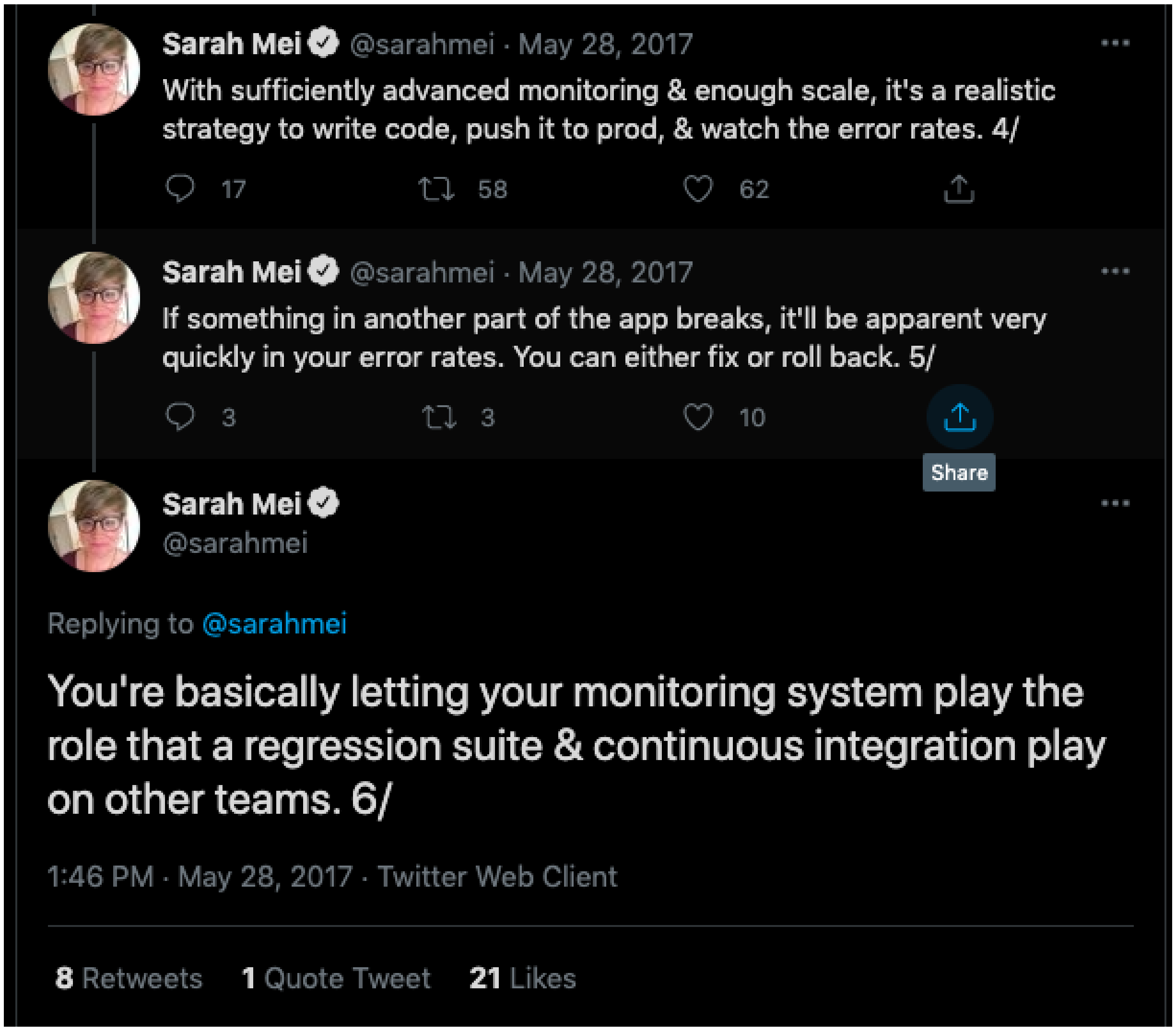

CODE-DRIVEN TESTING
REMANIEMENT DE CODE
1
(Ré)écrire le test.
Le test échoue.
Le test passe!
Apporter un changement minimal au code.
3
Vérifier que le test échoue.
2
Le test échoue.
Le test passe.
Les tests passent.
Remanier le code.
5
4
Vérifier que tous les tests passent.
5'
Mettre à jour les tests en échec.
Des tests échouent.
Corriger les régressions.
La qualité du code est jugée satisfaisante.
Itérer
focus
Réalisation du contrat défini par le test
focus
Réalignement de la conception avec les besoins connus
TEST-DRIVEN DEVELOPMENT
Xavier Pigeon

Testing In Production

The traditional view is that the goal of testing is to prevent shipping bugs into production. Therefore, by definition, you must do your testing offline before your system goes into production. However, there are two caveats:

1. Informal surveys reveal that the percentage of bugs found by automated tests is surprisingly low.
2. On top of that, modern service-oriented distributed systems (which are deployed in most software engineering organizations nowadays) are particularly hard to test. The interactions between the components can get tricky.

Here is our philosophy for testing in production: *Bugs are inevitable, so you might as well set up the system so that users can help you find them.*


Sarah Mei @sarahmei · May 28, 2017

With sufficiently advanced monitoring & enough scale, it's a realistic strategy to write code, push it to prod, & watch the error rates. 4/

17 58 62

Sarah Mei @sarahmei · May 28, 2017

If something in another part of the app breaks, it'll be apparent very quickly in your error rates. You can either fix or roll back. 5/

3 3 10

Sarah Mei
@sarahmei

Replying to @sarahmei

You're basically letting your monitoring system play the role that a regression suite & continuous integration play on other teams. 6/

1:46 PM · May 28, 2017 · Twitter Web Client

8 Retweets 1 Quote Tweet 21 Likes

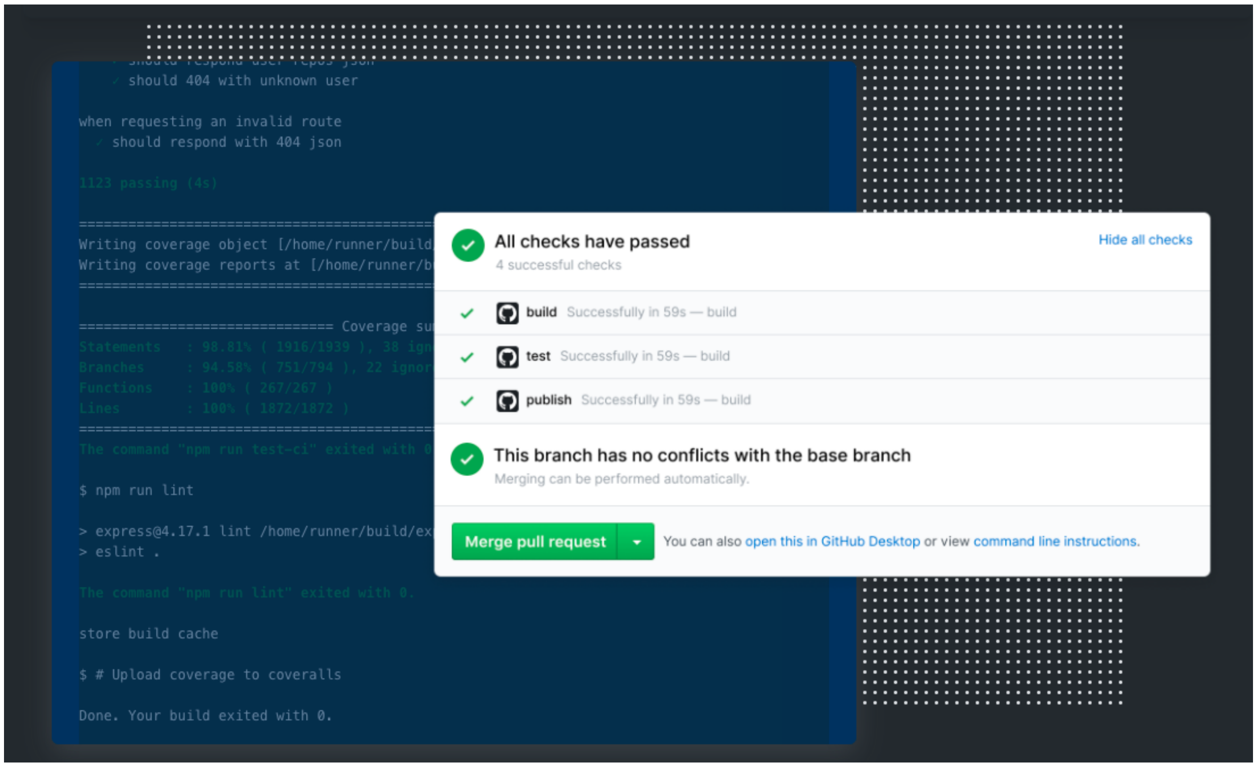
There are a few strategies to test in production:

1. **Canary deployment**: Do not roll out the new software version to all the users right away. Instead, just roll it out to a small percentage of your users and separately monitor that group's behavior.
2. **A/B testing:** You can run a more principled statistical test if you have particular metrics that you care about: one for the old version of the code that is currently running and another for the new version that you are trying to test.
3. **Real user monitoring:** Rather than looking at aggregate metrics (i.e., click-through rate), try to follow the journey that an actual user takes through your application and build a sense of how users experience the changes.
4. **Exploratory testing:** Testing in production is not something that you want to automate fully. It should involve a bit of exploration (individual users or granular metrics).

Continuous Integration and Continuous Delivery

CI/CD platforms automate the tests that you run by hooking into your code repo. When you trigger some actions to take place (pushing new code, merging new code into a branch, submitting a pull request), CI/CD platforms kick off a job that is responsible for packaging your code, running all your tests, producing a report that tells you how well your code performs on your tests, and gatekeeping whether your new code can make it to the next stage. Tactically, you can define these jobs as commands in a Docker container and store the results for later review.

SaaS solutions for continuous integration include CircleCI and Travis CI. Most of them do not have GPUs available. If you are just getting started, the default recommendation is GitHub Actions, which is super easy to integrate.



```
      ✓ should respond user repos json
      ✓ should 404 with unknown user

    when requesting an invalid route
      ✓ should respond with 404 json

  1123 passing (4s)

=============================================
Writing coverage object [/home/runner/build
Writing coverage reports at [/home/runner/b
=============================================

============================== Coverage su
Statements   : 98.81% ( 1916/1939 ), 38 ign
Branches     : 94.58% ( 751/794 ), 22 ignor
Functions    : 100% ( 267/267 )
Lines        : 100% ( 1872/1872 )
=============================================
The command "npm run test-ci" exited with 0

$ npm run lint

> express@4.17.1 lint /home/runner/build/ex
> eslint .

The command "npm run lint" exited with 0.

store build cache

$ # Upload coverage to coveralls

Done. Your build exited with 0.
```

All checks have passed
4 successful checks
Hide all checks

✓ build Successfully in 59s — build
✓ test Successfully in 59s — build
✓ publish Successfully in 59s — build

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request
You can also open this in GitHub Desktop or view command line instructions.

Jenkins and Buildkite are manual options for running continuous integration on your own hardware, in the cloud, or something in between. There is a lot more flexibility about the types of jobs you can run through the systems (meaning you can use your GPUs). The tradeoff is that they are harder to set up.

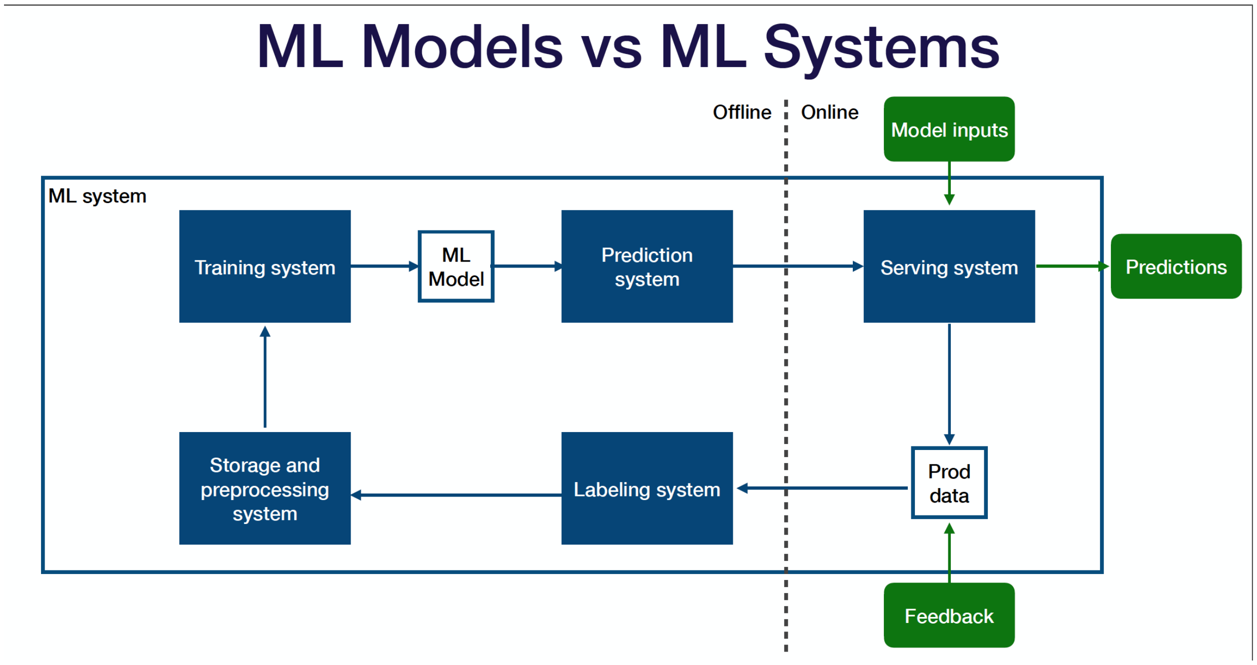
3 - Testing Machine Learning Systems

There are several **core differences** between traditional software systems and ML systems that add complexity to testing ML systems:

- Software consists of only code, but ML combines code and data.
- Software is written by humans to solve a problem, while ML is compiled by optimizers to satisfy a proxy metric.
- Software is prone to loud failures, while ML is prone to silent failures.
- Software tends to be relatively static (in principle), while ML is constantly changing.

Due to such differences, here are **common mistakes** that teams make while testing ML systems:

- Think the ML system is just a model and only test that model.
- Not test the data.
- Not build a granular enough understanding of the performance of the model before deploying it.
- Not measure the relationship between model performance metrics and business metrics.
- Rely too much on automated testing.
- Think offline testing is enough, and therefore, not monitor or test in production.



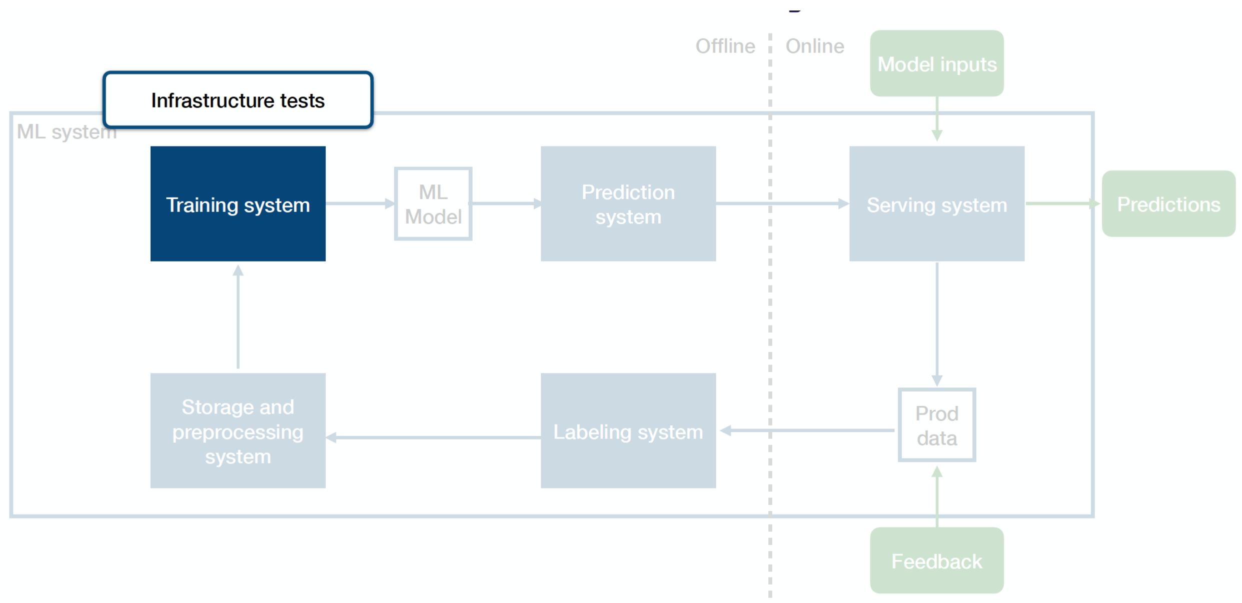
Above is the diagram of how you can think of your entire production ML system that straddles across the offline and online environments:

- Sitting in the middle is your **ML model** - an artifact created by your training process, which takes in an input and produces an output.
- The **training system** takes code and data as inputs and produces the trained model as the output.
- The **prediction system** takes in and pre-processes the raw data, loads the trained ML model, loads the model weights, calls *model.predict()* on the data, post-processes the outputs, and returns the predictions.

- Once you deploy your prediction system to the online environment, the **serving system** takes in requests from users, scales up and down to meet the traffic demands, and produces predictions back to those users.
- The whole ML system closes the loop by collecting **production data** (both the predictions that the model produces and additional feedback from users, business metrics, or labelers) and sending them back to the offline environment.
- The **labeling system** takes the raw data seen in production, helps you get inputs from labelers, and provides labels for that data.
- The **storage and pre-processing system** stores and pre-processes the labeled data before passing it back to the training system.

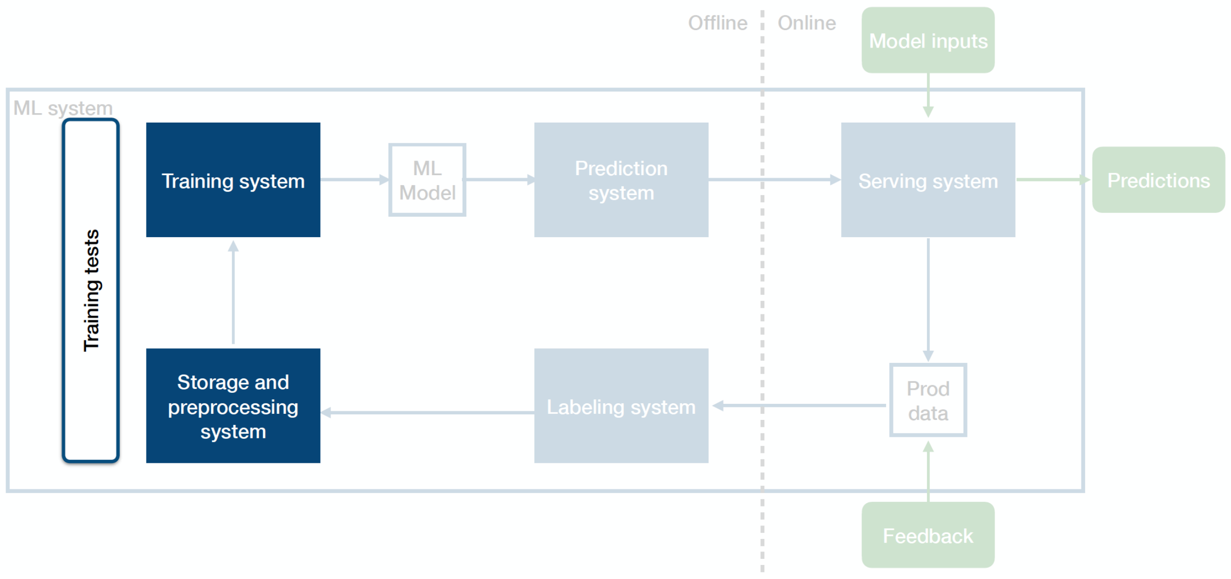
One way to think about how to test ML systems the right way is to **think about the tests that you can run for each system component and across the border of these components.**

Infrastructure Tests



Infrastructure tests are unit tests for your training system. They help you avoid bugs in the training pipeline. You can unit test your training code like any other code. Another common practice is to add single-batch or single-epoch tests that check performance after an abbreviated training run on a tiny dataset, which catches obvious regressions to your training code. Tactically, you should run infrastructure tests frequently during the development process.

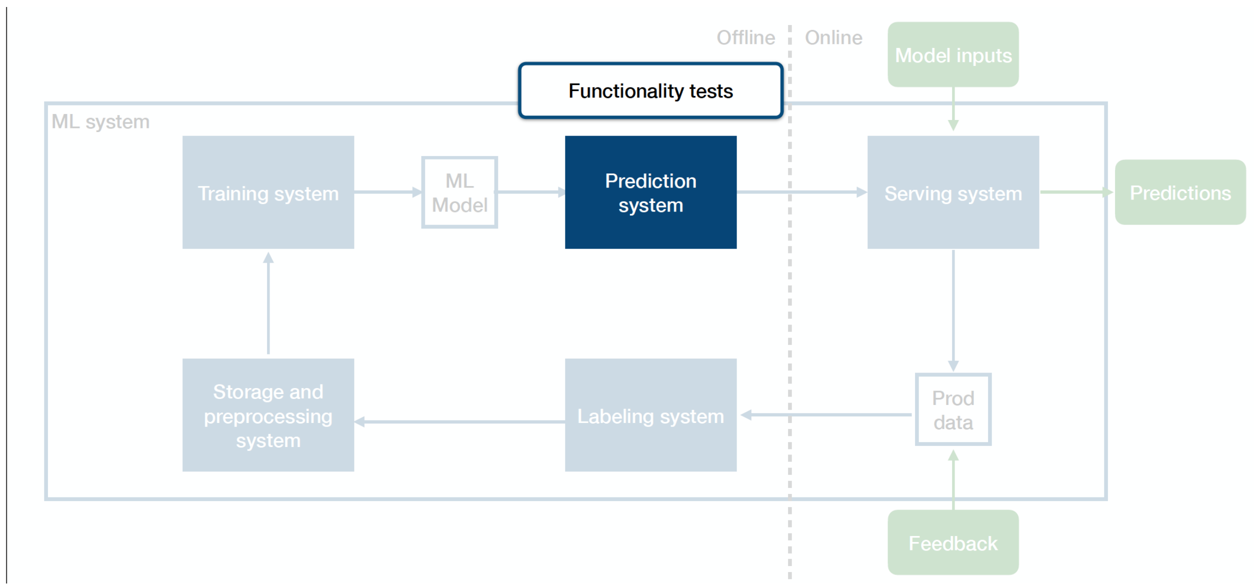
Training Tests



Training tests are integration tests between your data system and your training system. They make sure that training jobs are reproducible.

- You can pull a fixed dataset and run a full or abbreviated training run on it. Then, you want to check and ensure that the model performance on the newly trained model remains consistent with the reference performance.
- Another option is to **pull a sliding window of data** (maybe a new window for every few days) and run training tests on that window.
- Tactically, you should run training tests periodically, ideally nightly for frequently changing codebase.

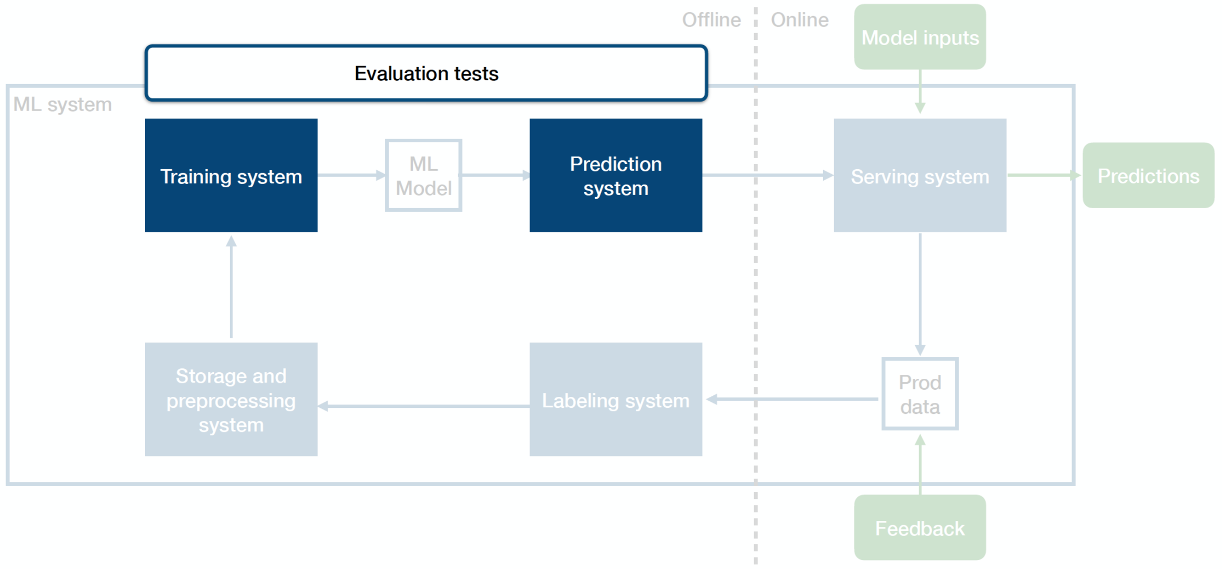
Functionality Tests



Functionality tests are unit tests for your prediction system. They help you avoid regressions in code that makes up your prediction infrastructure.

- You can unit test your prediction code like any other code.
- Specifically for the ML system, you can **load a pre-trained model and test its predictions on a few key examples**.
- Tactically, you should run functionality tests frequently during the development process.

Evaluation Tests



Evaluation tests are integration tests between your training system and your prediction system. They make sure that a newly trained model is ready to go into production. These make up the bulk of what's unique about testing ML systems.

- At a high level, you want to evaluate your model on all of the **metrics**, **datasets**, and **slices** that you care about.
- Then, you want to compare the new model to the old and baseline models.
- Finally, you want to understand the **performance envelope** of the new model.
- Operationally, you should run evaluation tests whenever you have a new candidate model considered for production.

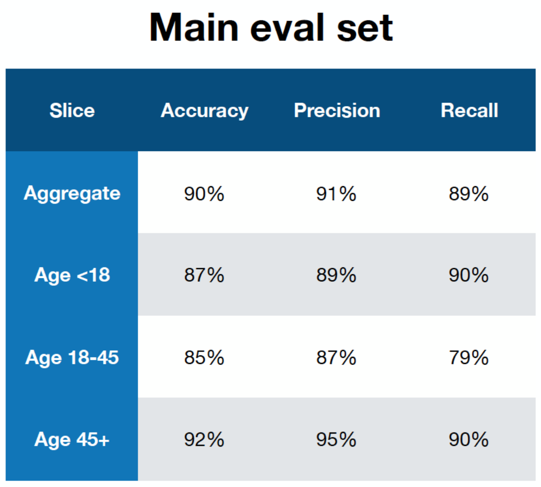
It is important to note that evaluation tests are more than just the validation score. They look at **all the metrics that you care about**:

- **Model metrics**: precision, recall, accuracy, L2, etc.
- **Behavioral metrics**: The goal of behavioral tests is to ensure the model has the invariances we expect. There are three types of behavioral tests: (1) *invariance tests* to assert that the change in inputs shouldn't affect outputs, (2) *directional tests* to assert that the change in inputs should affect outputs, and (3) *minimum functionality tests* to ensure that certain inputs and outputs should always produce a given result. Behavioral testing metrics are primarily used in NLP applications and proposed in the Beyond Accuracy paper by Ribeiro et al. (2020).
- **Robustness metrics**: The goal of robustness tests is to understand the model's performance envelope (i.e., where you should expect the model to fail). You can examine feature importance, sensitivity to staleness, sensitivity to data drift, and correlation between model performance and business metrics. In general, robustness tests are still under-rated.
- **Privacy and fairness metrics**: The goal of privacy and fairness tests is to distinguish whether your model might be biased against specific classes. Helpful resources are Google's Fairness Indicators and the Fairness Definitions Explained paper by Verma and Rubin (2018).
- **Simulation metrics**: The goal of simulation tests is to understand how the model performance could affect the rest of the system. These are useful when your model affects the real world (for systems such as autonomous vehicles, robotics, recommendation systems, etc.). Simulation tests are hard to do well because they require a model of how the world works and a dataset of different scenarios.

Instead of simply evaluating the aforementioned metrics on your entire dataset in aggregate, you should also evaluate these metrics **on multiple slices of data**. A slice is a mapping of your data to a specific category. A natural question that arises is how to pick those slices. Tools like What-If and SliceFinder help surface the slices where the model performance might be of particular interest.

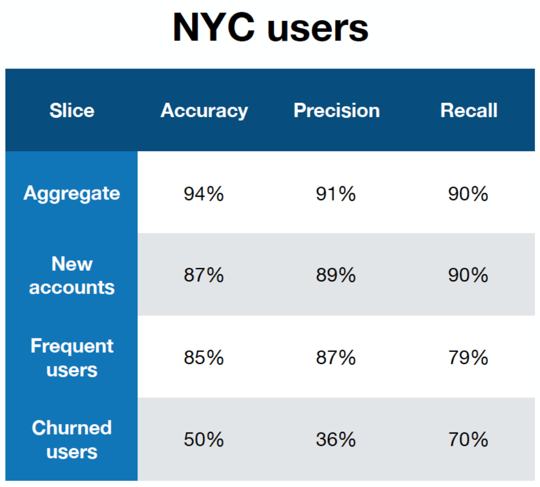
Finally, evaluation tests help you **maintain evaluation datasets for all of the distinct data distributions you need to measure.** Your main validation or test set should mirror your production distribution as closely as possible as a matter of principle. When should you add new evaluation datasets?

- When you collect datasets to specify specific edge cases.
- When you run your production model on multiple data modalities.
- When you augment your training set with data not found in production (synthetic data).



Main eval set

Slice	Accuracy	Precision	Recall
Aggregate	90%	91%	89%
Age <18	87%	89%	90%
Age 18-45	85%	87%	79%
Age 45+	92%	95%	90%



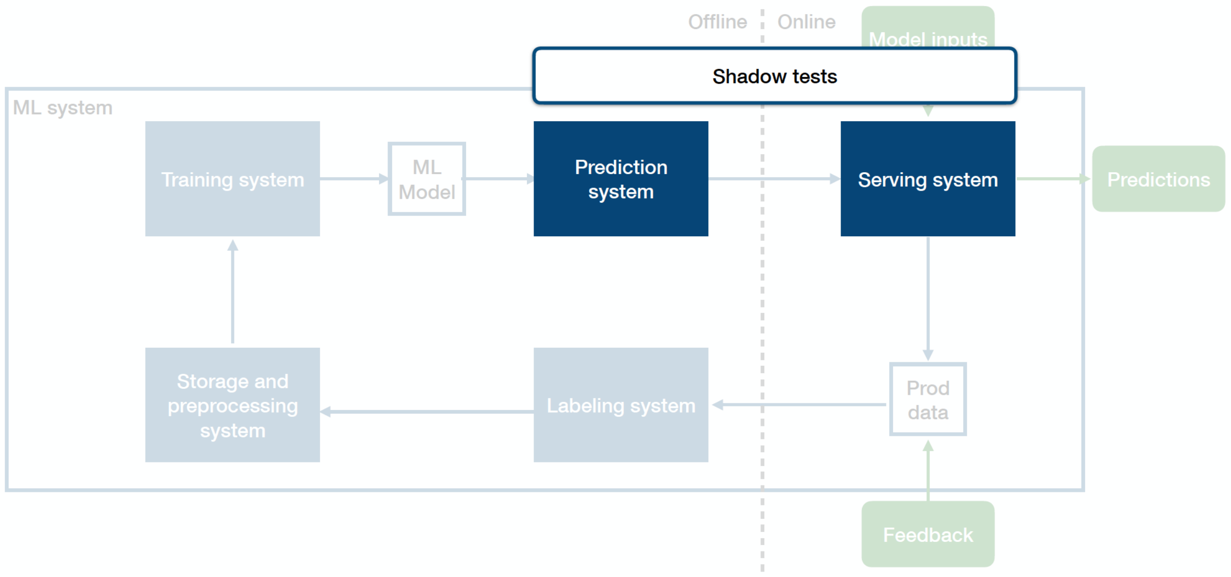
NYC users

Slice	Accuracy	Precision	Recall
Aggregate	94%	91%	90%
New accounts	87%	89%	90%
Frequent users	85%	87%	79%
Churned users	50%	36%	70%

The report produced by the evaluation system entails the metrics broken down against each of the data slices. How can you decide whether the evaluation passes or fails?

At a high level, you want to compare the new model to **the previous model** and **another fixed older model**. Tactically, you can (1) set thresholds on the differences between the new and the old models for most metrics, (2) set thresholds on the differences between data slices, and (3) set thresholds against the fixed older model to prevent slower performance "leaks."

Shadow Tests



Shadow tests are integration tests between your prediction system and your serving system. They help you catch production bugs before those bugs meet users. In many settings, models (which are built in frameworks such as sklearn, Pytorch, TensorFlow, etc.) are developed in isolation from the existing software system. For example, a model to flag inappropriate tweets may be developed in TensorFlow on a static set of data, not directly in the streaming environment of the broader software architecture. Because the prediction system and the serving system are developed in different settings with different assumptions and environments, there are many opportunities for bugs to creep in. These bugs can be tricky to catch prior to integration, so shadow tests can help identify them beforehand.

Firstly, shadow tests help you **detect bugs in the production deployment**. In the code path you're using to build the production model, maybe there's some bug there. You want to make sure that you catch that before users see that bug.

Secondly, shadow tests also help you **detect inconsistencies between the offline model and the online model**. There’s a translation step in the training pipeline in many companies - going from the offline trained model to the online production model (the model itself, the preprocessing pipeline, etc.). A common bug source in production ML systems happens because of the inconsistencies cropping up in that translation step. A good health check ensures that your actual production model is producing the exact predictions on a fixed set of data as the model you have running on your laptop.

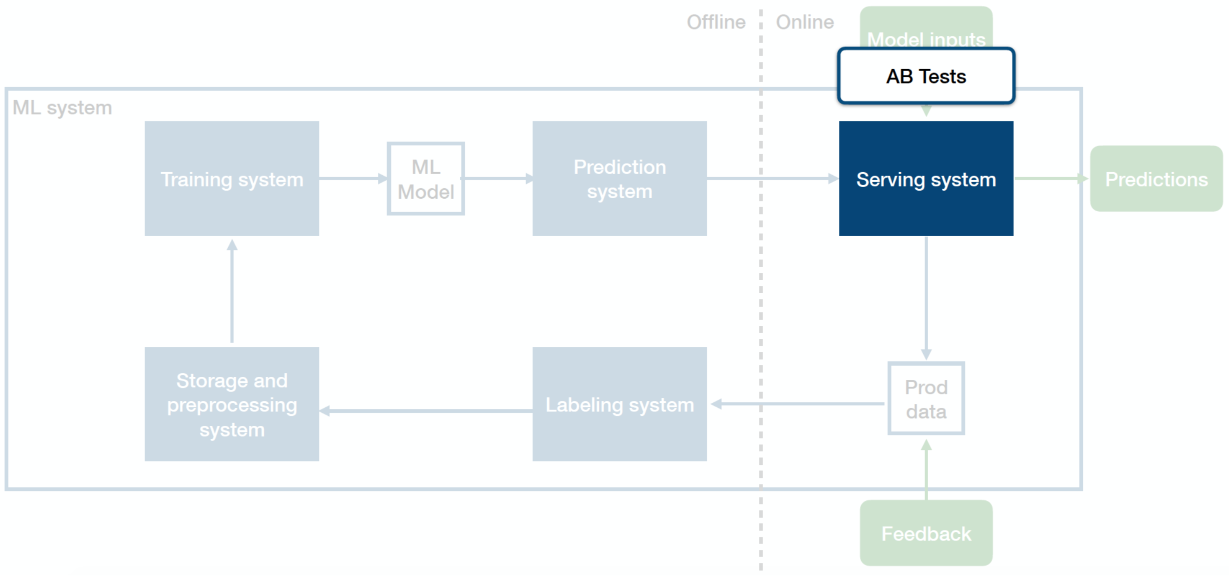
Thirdly, shadow tests help you **detect issues that don't appear on the data you have offline but appear on production data**.

How do we design shadow tests? These can require a significant amount of infrastructure, as they are dependent on actual model integration opportunities being available.

- Typical shadow tests involve **testing the performance of a candidate model on real data without returning or acting on the output**. For example, a company may integrate and run a new model alongside the previous model without returning the output to the user.
- **Analyzing the consistency of the predictions between the two models** can help spot important differences before they impact production performance.
- Another option is to gather production data, save it offline, and **test the model's performance on the fresh data offline**.

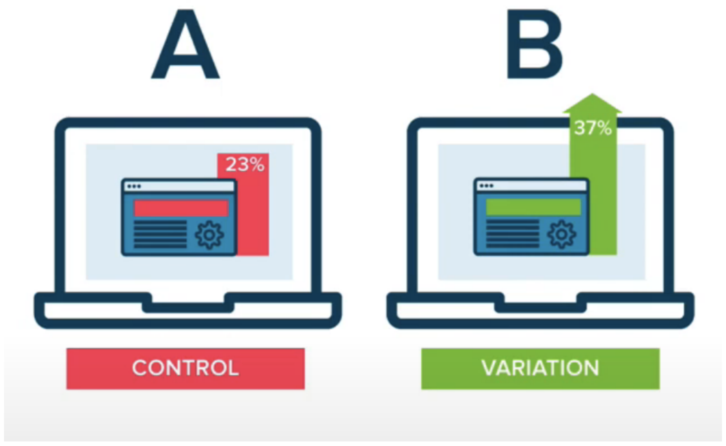
Overall, evaluating the distribution of model predictions in offline vs. online settings, candidate vs. production, or any similar setting of a model update before deploying a new model can help you avoid bugs.

A/B Tests



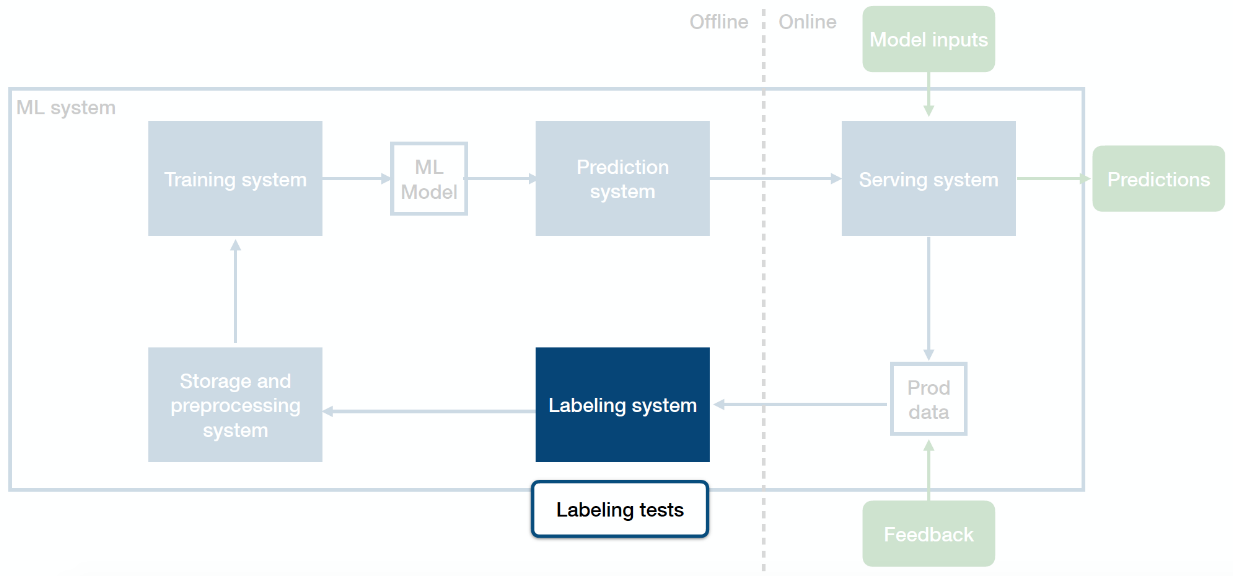
Shadow tests evaluate the prediction performance of a model as part of the broader software architecture, but not **the impact on users**. **A/B tests fill this role**. A/B tests are a common practice in software engineering, especially in web systems. A/B testing is defined as "a randomized experimentation process wherein two or more versions of a variable (web page, page element, etc.) are shown to different segments of website visitors at the same time to determine which version leaves the maximum impact and drive business metrics."[1]

[1] https://vwo.com/ab-testing-2/



In model evaluation, A/B tests determine the impact of different model predictions on user and business metrics. One common way of A/B testing models is **to "canary" data** or return predictions on a small portion of the data (i.e., 1% or 10%) to the relevant users. The remaining data acts as a control and functions under existing system behavior (i.e., an old model or even no model). Evaluating the difference in metrics between the two groups can determine the relative impact of your model. This simple baseline can work well. **Adding more statistically principled splits**, which is common in A/B testing, can be a good idea.

Labeling Tests

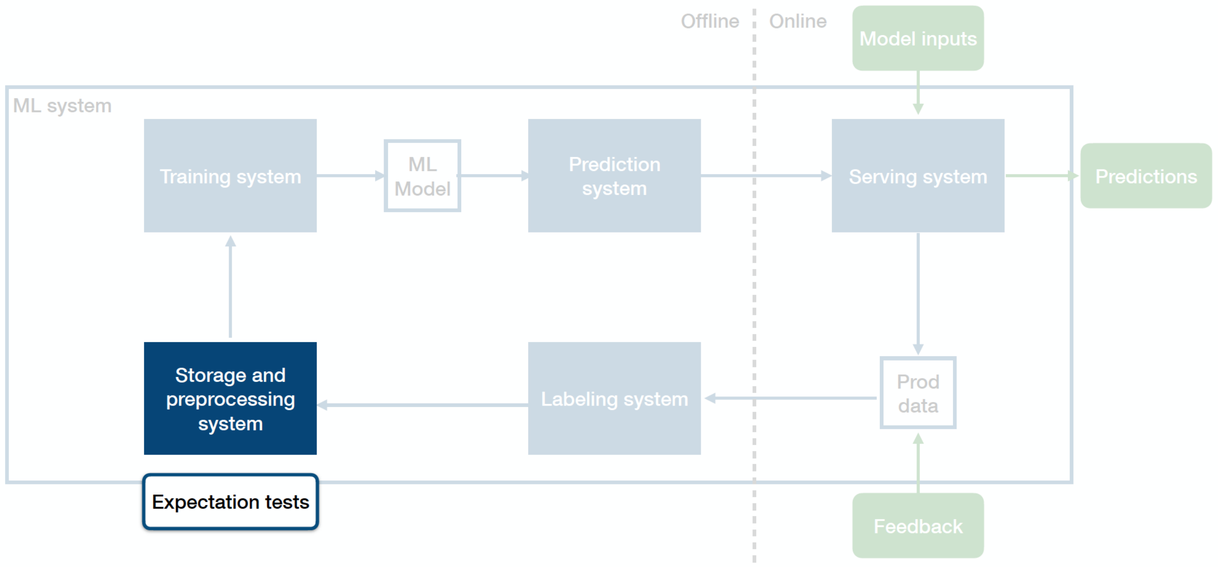


Machine learning models operate in a GIGO paradigm: garbage in, garbage out. **To prevent poor quality labels from cropping up and corrupting the model, you need to unit test the labeling systems and procedures.**

- You should start by training, certifying, and evaluating individual labelers, who each play a crucial role in the quality of the labels.

- A simple and common label quality test is **to spot check labels as they come in** by opening up 100 or 1000 labels from a batch and evaluating them yourself to understand their quality. Using a performant model's guidance, you can make this process more efficient and only look at labels where the model and the labeler disagree.
- Another test can be to **aggregate labels of multiple labels and measure agreement across labels**. The higher the agreement, the better quality the labels are.
- Using metrics of agreement, you can assign "trust scores" to labelers based on their performance relative to other labelers and weigh the labels accordingly.

Expectation Tests



Expectation tests address the data preprocessing and storage system. Essentially, they are unit tests for your data. They are designed to catch data quality issues and bad data before they make their way into the pipeline.

The typical way that expectation tests operate is rule- or threshold-based. At each step of the data processing pipeline, the output should conform to a specific format that matches a rule or specific format. If the rule or threshold does not pass, then that stage of the expectation test and the data pipeline's related step fails. Such tests are frequently run with batch data pipeline jobs. Great Expectations is an open-source library gaining popularity for running expectation tests. The library allows you to set hard rules for the kinds of values or behaviors (i.e., statistics) you expect from your data.

How do you set the rules and thresholds for expectation tests? Most expectation tests are set manually. A more sophisticated option is to profile a high-quality sample of your data and set thresholds accordingly. In practice, to avoid false alarms from overly sensitive tests, a combination of both approaches is needed.

Challenges and Recommendations Operationalizing ML Tests

Running tests is an excellent idea in theory but can pose many practical challenges for data science and ML teams.

- The first challenge is often **organizational**. In contrast to software engineering teams for whom testing is table stakes, data science teams often struggle to implement testing and code review norms.
- The second challenge is **infrastructural.** Most CI/CD platforms don't support GPUs, data integrations, or other required elements of testing ML systems effectively or efficiently.
- The third challenge is **tooling**, which has not yet been standardized for operations like comparing model performance and slicing datasets.
- Finally, **decision-making** for ML test performance is hard. What is "good enough" test performance is often highly contextual, which is a challenge that varies across ML systems and teams.

Let's boil all these lessons for testing down into **a clear set of recommendations** specific to ML systems:

1. Test each part of the ML system, not just the model. You build the machine that builds the model, not just the model!
2. Test code, data, and model performance, not just code.
3. Testing model performance is an art, not a science. There is a considerable amount of intuition that guides testing ML systems.
4. Thus, the *fundamental* goal of testing model performance is to **build a granular understanding** of how well your model performs and where you don't expect it to perform well. Using this intuition derived from testing, you can make better decisions about productionizing your model effectively.
5. Build up to this gradually! You don't need to do everything detailed in this lecture, and certainly not all at once. Start with:
    a. Infrastructure tests
    b. Evaluation tests
    c. Expectation tests

4 - Explainable and Interpretable AI

Definitions

What do explainable and interpretable AI, buzzwords you've undoubtedly heard before, actually mean? Let's start by outlining some more fundamental terms about the problem space:

- **Domain predictability**: the degree to which it is possible to detect data outside the model's domain of competence.
- **Interpretability**: the degree to which a human can consistently predict the model's result (Kim et al., 2016).
- **Explainability**: the degree to which a human can understand the cause of a decision (Miller, 2017).

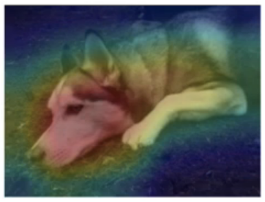
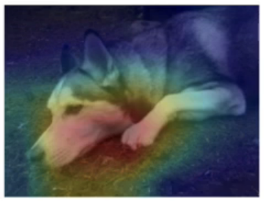
We'll walk through four different methods of making models interpretable and explainable:

1. Use an interpretable family of models.
2. Distill the complex model to an interpretable one.
3. Understand the contribution of features to the prediction.
4. Understand the contribution of training data points to the prediction.

Use An Interpretable Family of Models

Examples of interpretable families of models are **simple, familiar models like linear regression, logistic regression, generalized linear models, and decision trees**. If you understand the math of these models, it's pretty easy to understand why a model made the decision it did. Because of the reasonably elementary math, these models are interpretable and explainable. However, they are not very powerful.

Another class of models that are interpretable is **attention models**. Examining where a model is "looking" helps us anticipate a model's prediction, thus making them interpretable. However, attention maps are not particularly explainable. They do not produce *complete* explanations for a model's output, just a directional explanation. Furthermore, attention maps are not reliable explanations. Attention maps tell us only where a model is looking, not why it is looking there. Frequently, models focus exclusively on an image's salient region without an underlying reasoning that relates to the task at hand. In the sample below, the attention model is "looking" at the salient region for classification, which has a very different meaning in each context.

	Test Image	Evidence for Animal Being a Siberian Husky	Evidence for Animal Being a Transverse Flute
Explanations Using Attention Maps			

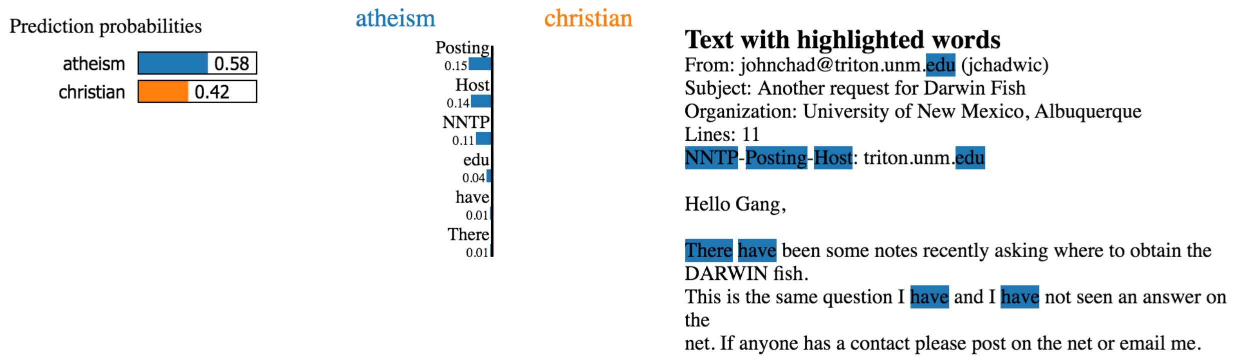
The conflation of attention with explanation is a critical pitfall to avoid.

Distill A Complex To An Interpretable One

Instead of restricting models to only interpretable families, we can fit a more complex model and interpret its decision using another model from an interpretable family. The trick is to train this additional model, referred to as **a surrogate model**, on the raw data and the complex model's predictions. The surrogate model's corresponding interpretation can be used as a proxy for understanding the complex model.

This technique is quite simple and fairly general to apply. In practice, however, two concerns manifest.

1. If the surrogate itself performs well on the predictions, why not try to directly apply it rather than the more complex model?
2. If it doesn't perform well, how do we know that it genuinely represents the complex model's behavior?



Text with highlighted words
From: johnchad@triton.unm.edu (jchadwic)
Subject: Another request for Darwin Fish
Organization: University of New Mexico, Albuquerque
Lines: 11
NNTP-Posting-Host: triton.unm.edu

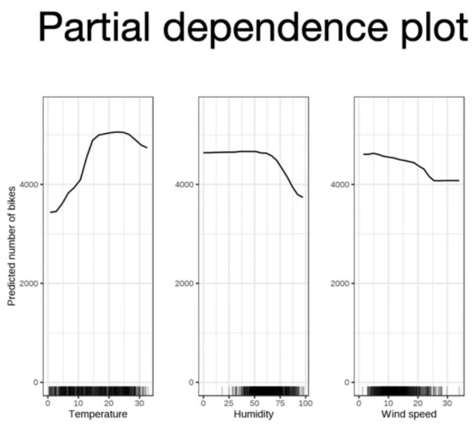
Hello Gang,

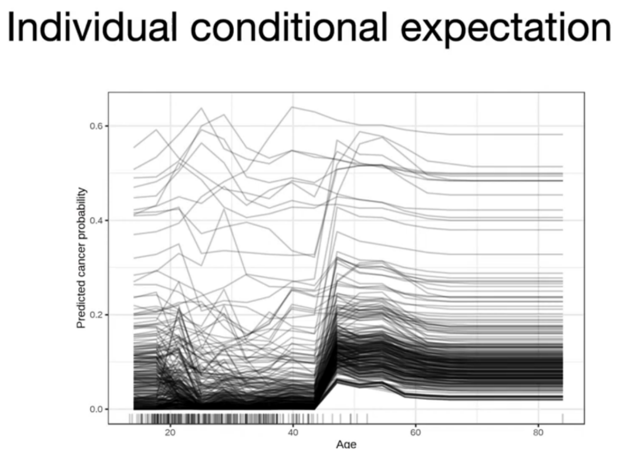
There have been some notes recently asking where to obtain the DARWIN fish.
This is the same question I have and I have not seen an answer on the
net. If anyone has a contact please post on the net or email me.

Another category of surrogate models is local surrogate models **(LIME)**. Rather than apply the surrogate model in a global context on all the data, LIME models focus on a single point to generate an explanation for. A perturbation is applied to the point, resulting in a local neighborhood of perturbed data points. On top of these perturbed data points, a surrogate model is trained to map the points to the original predictions from the complex model. If the surrogate model classifies similarly to the complex model, the surrogate can be considered a proxy for interpretation purposes. This method is used widely, as it works for all data types (including images and text). However, defining the right perturbations and ensuring the stability of the explanations is challenging.

Understand The Contribution of Features To The Prediction

Better understanding each feature's role in making a prediction is another option for interpretable and explainable ML. **Data visualization** is one such option, with plots like partial dependence plots and individual conditional expectation plots.



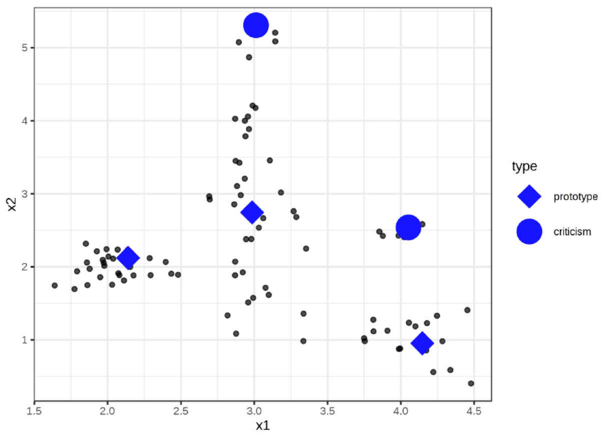


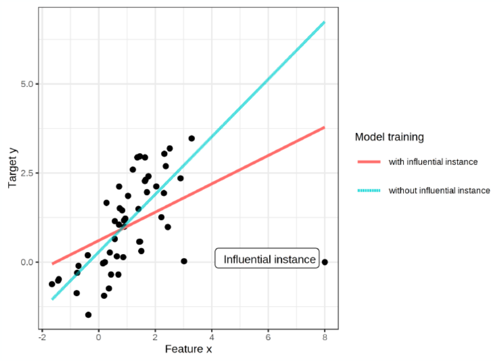
A numerical method is **permutation feature importance**, which selects a feature, randomizes its order in the dataset, and sees how that affects performance. While this method is very easy and widely used, it doesn't work for high-dimensional data or cases where there is feature interdependence.

A more principled approach to explaining the contribution of individual features is **SHAP** (Shapley Additive Explanations). At a high level, SHAP scores test how much changes in a single feature impact the output of a classifier when controlling for the values of the other features. This is a reliable method to apply, as it works on a variety of data and is mathematically principled. However, it can be tricky to implement and doesn't provide explanations.

Gradient-based saliency maps are a popular method for explanations and interpretations. This intuitive method selects an input, performs a forward pass, computes the gradient with respect to the pixels, and visualizes the gradients. Essentially, how much does a unit change in the value of the input's pixels affect the prediction of the model? This is a straightforward and common method. Similar to the challenge with attention, the explanations may not be correct, and the overall method is fragile and sensitive to small changes.

Understand The Contribution of Training Data Points To The Prediction





Instead of focusing on features and their explicit relevance to the prediction, we can also take a hard look at the training data points themselves.

- **Prototypes and criticisms** are one such approach, though it is less applicable to deep learning. In this method, prototypes are clusters of data that explain much of the variance in the model. Criticisms are data points not explained by the prototypes.
- Another approach is to look specifically at "**influential instances**" or data points that cause major changes in the model's predictions when removed from the data set.

Do You Need "Explainability"?

A good question to ask yourself whether or not "explainable AI" is a real need for your applications. There are a couple of cases where this question can be useful:

1. **Regulators demand it.** In this case, there's not much you can do besides produce some kind of explainable model. However, it can be helpful to **ask for clarification** on what explainability is judged as.
2. **Users demand it.** In some cases, users themselves may want trust or explainability in the system. Investigate the necessity for the explainability and trust to come directly from the model itself. **Can good product design inspire trust more effectively?** For example, allowing doctors to simply override models can reduce the immediate need for explainability. A big associated concern is how often users interact with the model. Infrequent interactions likely require explainable AI, as humans do not get a chance to build their feel for the system. More frequent interactions allow for the simpler objective of interpretability.
3. **Deployment demands it.** Sometimes, ML stakeholders may demand explainability as a component of ensuring confidence in ML system deployment. In this context, explainability is the wrong objective; **domain predictability is the real aim**. Rather than

full-on explainability, interpretability can be helpful for deployment, especially visualizations for debugging.

At present, true explainability for deep learning models is not possible.

- Current explanation methods are not faithful to the original model performance; it can be easy to cherry-pick specific examples that can overstate explainability.
- Furthermore, these methods tend to be unreliable and highly sensitive to the input.
- Finally, as described in the attention section, the full explanation is often not available to modern explainability methods.

Because of these reasons, explainability is not practically feasible for deep learning models (as of 2021). Read Cynthia Rudin's 2019 paper for more detail.

Caveats For Explainable and Interpretable AI

- *If you genuinely need to explain your model's predictions, use an interpretable model family (read more here).*
- *Don't try to force-fit deep learning explainability methods; they produce cool results but are not reliable enough for production use cases.*
- *Specific interpretability methods like LIME and SHAP are instrumental in helping users reach interpretability thresholds faster.*
- *Finally, the visualization for interpretability can be pretty useful for debugging.*

5 - Resources

- ML Test Score Paper
- Behavioral testing paper
- Jeremy Jordan's effective testing
- Robustness Gym
- Made with ML's guide to testing
- Eugene Yan's practical guide to maintaining machine learning
- Chip Huyen's CS329 lecture on evaluating models
- Interpretable ML Book