Practical machine learning engineering

Curated guide for building and maintaining solid end-to-end machine learning pipelines

Author: Morales, J.A.
Last update: January 13, 2023

Contents

Introduction

Curated guide for building and maintaining solid end-to-end machine learning pipelines.

Do we really need machine learning?

In building a machine learning pipeline, the first question we ask is, “do we really need one?” We want to focus on the problem we’re trying to solve, even if it’s simply wanting to explore a solution or create a proof-of-concept.

Start measuring before building

There are many reasons for doing this. It provides additional information for decisions along the build process. It also provides a pulse on the data – it’ll help capture any changes before, during, and after engineering process.

How to know if you’re ready for machine learning

One sign the project is ready for machine learning is when heuristics begin to get too complex. Simple heuristics work but complicated heuristics can get messy and will hurt more in the long run. At this point, consider machine learning instead. The trifecta you want: unwieldy heuristics + enough data + clear objective = start machine learning.

Opt for a simple model, with simple features

Keeping the first model simple allows you to more efficiently get the infrastructure right. It makes it easier to build up, maintain, and troubleshoot the data and the model.

At any point, be clear about what you’re testing

Test the model, infrastructure, and their integration separately. This helps isolate problems as well as package, prioritize, and plan work.

Don’t drop the data, unless noted

It’s not uncommon to duplicate an existing pipeline as a starting point for a new build. Caveat: the old pipeline might have dropped data that we need in the new pipeline. Make sure it’s noted. This also applies to any filtering and significant data transformations.

Heuristics can still be handy

If we have pre-existing heuristics that already provide good insight or could potentially help us understand better, we can try to use them by turning them into features.

Practice ‘alerting hygiene’ to keep it fresh

Get the model out fast – and keep it fresh. Know the freshness requirements of the system and build them into actionable alerts and dashboards.

Detect the defects or don’t serve

It’s obvious but it’s important to mention. Promote to higher environments only after testing, fixing, and/or noting bugs. Even if it’s a POC project. Yes, release that MVP. But. Don’t serve to the user if you haven’t done your detecting.

How to hear the silent failures

Compared to other system failures, silent errors are tricky to catch. Many mechanisms are in place to catch the loud and noticeable issues. To reduce silent failures however, it's helpful to keep statistics and manually inspect the data on occasion.

Who owns your features?

Give feature columns owners. In development, we often hear that the documentation is in the code. However, it can be time consuming to piece together the undisclosed or changing significance of data. At critical failure points, in production, this could make or break the solution.

All the objectives in the world

It’s great to check all the objectives but sometimes it’s impractical to work through them all. One way to prioritize is to look at what’s optimizable. We can work to increase this score.

How to choose proxy objectives

What’s your criteria for measuring proxy objectives – the things that we can directly evaluate in lieu of the real objectives? Ideally, proxy objectives are simple, observable, and attributable.

Transparent choices mean debugging made easy

Starting with an interpretable model makes troubleshooting easier. We can re-trace our steps and understand differences more clearly.

Do measures have the same assumptions?

For example, quality ranking assumes that data is posted in good faith. However, spam-filtering assumes that some aren’t. We can arrange the tasks so that they strengthen each other’s purpose. More importantly, we want to be clear about the assumptions. At some point, we may need to call on these assumptions to make important decisions.

Fail fast then iterate to fail less and faster

Plan to launch quickly. Try not to add things that will slow down early and future releases.

Start with the directly observable features

Instead of relying early on learned features from the get go, begin by looking at the directly observed and reported features. They’re simpler, faster to implement, and easier to troubleshoot.

Compare measures with existing measures

If the model is going to be a part of a family of systems, can we leverage the other systems to check the goodness of the model, the data, and the hypotheses?

Many but simple vs few but complex features

If possible, use very specific features. You can aggregate later. Normalize now.

If you must aggregate the features

Combine and modify existing features to create new features in human-understandable ways. Don't forget to note the transformation that you made.

Number of features is proportional to the number of examples

The number of feature weights you can learn in a linear model is roughly proportional to the amount of data you have.

Don't be afraid to drop features

Clean up features that you're not using anymore. Unused features create technical debt.

Test as the user, not only as the coder

Fishfooding is when you use a prototype within your team. Dogfooding is when you use a prototype within your company. Both test from the coder's perspective. Create user personas and get them to test early.

Measure between models

Measure your model against production, against other prototypes, and itself.

Stability over predictive powers

Choose function and usefulness over bells and whistles.

Patterns everywhere

Look for patterns in the measured errors. Are there trends outside the current set of features?

Quantify undesirable behaviour

What to do with undesirable behaviours: spot, observe, quantify, and address. Undesirable behaviours in the system include pet peeves. It's good to measure these so they can be used to your systems advantage. You can turn them into features, objectives, or find a way to remove them from your system.

Short-term behaviour doesn't necessarily extend to long-term

Be aware that identical short-term behaviour doesn't imply identical long-term behaviour. If the model behaves differently from before, suspect possible sources of change such as data or data-handling.

Save the training features in a log

One way to make sure that you train like you serve is to save the set of features used at serving time. Put those features on a log and leverage them during training time. It doesn't have to be for every example. Even a small fraction might help verify consistency between serving and training data.

Sample by importance

It's ideal not to arbitrarily drop data. Instead, choose by importance. Get more details where it matters.

Stability over predictive powers

Choose function and usefulness over bells and whistles.

Sources can change anytime

If you source data from a seemingly stable table, remember that even that data could change. Data could change before, during, and after deployment. Connections to that data such as joins could get affected.

Re-use code

Some differences arise between the time of building and the time the model goes live. There are a few things that we can try to keep consistent, like the code.

Test beyond the deadline

If you produce a model based on the data until January 5th, test the model on the data on January 6th and after. To get a better reflection of how the system will do in production, test the model on the data gathered after the data you trained on.

Prioritize clean data over performance

In binary classification for filtering (e.g., spam detection or determining interesting emails), make small short-term sacrifices in performance for clean data. If you have to choose, bite the bullet and sacrifice a little bit of the performance for very clean data.

There may be skews

Beware of the inherent skew in problems like ranking. Design the model so that it can handle future data changes.

Try to get closer to a complete picture

A feature that may significantly impact performance is placement of a product. If you place a data product front and center in the user's ecosystem, it will likely get attention. Have a feature that captures the position of the product. This helps avoid confounding variables that puts you in feedback loops. Try to get as much of a complete picture as possible of the data and its context.

Measure before, during, and after build

There are several measures that may be worth keeping an eye on the difference between the performance on training data and holdout data, holdout data and next-day data, and next-day data and the live data.

Keep the objectives aligned

As measurements plateau, keep the objectives and the product goals in mind. If objectives are misaligned, we should be spending time on those and not on engineering new features. Is it the model or is it time to revisit the objectives and goals?

Decisions as proxies for goals

Launch decisions are not made solely on metrics under the control of the model. Some criteria we didn't measure or optimize in the model may matter more than the ones we did measure. It's possible for the decisions we make at launch time to be proxies for long-term product goals.

Separate base model from ensemble

Keep ensembles simple. One more way to keep things simple is to keep ensembles and base models separate. Each model can either be an ensemble only, which takes the input of other models only, or a base model, which takes in many raw features only.

When performance plateaus

When the measurements start to plateau, it may be time for an additional strategy. For example, start looking into qualitatively new sources and radically different features. At the same time, weigh the cost-benefit of the complexity before adding these features.

Results can sometimes be counter-intuitive

Initial expectations could easily get in the way of making the most of your model if you let them.

Don't expect patterns to be the same across domains

Even if it makes sense to do so, don't blindly expect patterns to be the same across domains, even if the domains overlap or are related to one another. You can use data from the different areas to confirm instead.

References

Buxton, B. (2010) *Sketching User Experiences: Getting the design right and the right design*. Morgan Kaufman.

Krug, S. (2013) *Don't make me think, revisited: A common sense approach to web usability*. 3rd edition. New Riders.

Zinkevich, M. (n.d.) 'Rules of machine learning: Best practices for ML engineering', Google. Available at: https://developers.google.com/machine-learning/guides/rules-of-ml/ (Accessed: 01 April 2019).