

STASH

ML Paper Group - Feb 15, 2019

[Hidden Technical Debt in Machine Learning Systems](#)

I. ML & tech debt

- technical debt = **long term costs** incurred by moving quickly
 - by-product of an **important tradeoff** in software development
- *“ML systems have a **special capacity** for incurring tech debt, because they have all of the maintenance problems of traditional code plus an additional set of **ML-specific issues**.”*
 - **“hidden” tech debt**
 - changes in data (eg generation or collection) are especially sneaky
 - treating models like black boxes can make assumptions hard to detect
- *“Not all debt is bad, but all debt needs to be serviced...**hidden debt is dangerous** because it compounds silently.”*

II. ML & abstraction boundaries

- traditional SE: encapsulation & modular design promote **abstraction boundaries**
 - easy to make isolated changes
 - crucial for scalability & ease of maintenance
- ML: external dependencies cause these boundaries to **erode**
 - ex: change in feature distribution (this is why we have to be vigilant about monitoring!)
 - ex: undeclared consumers (this problem arises with new event bus architecture!)

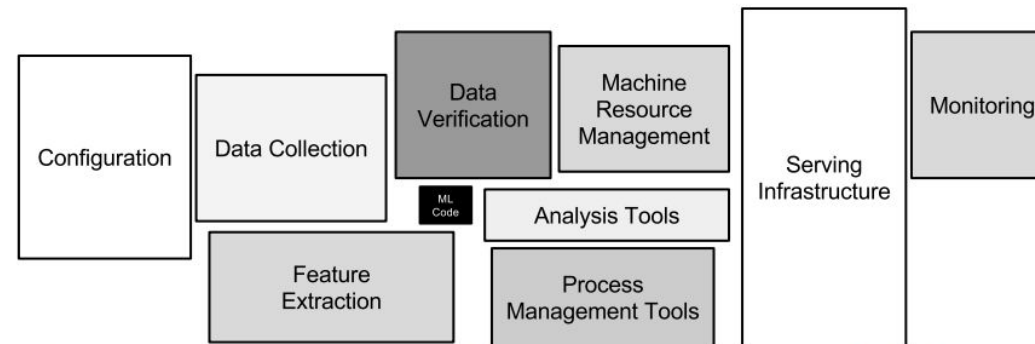


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

III. Sources of hidden tech debt: data

- **unstable data**: feature distributions can change for many reasons
 - data generating processes can change
 - data collection strategies can change
 - these changes are difficult to detect without proper monitoring
- **underutilized data**: yet another reason that fewer features are better
 - in addition to increasing model complexity, each feature increases your model's TCO
 - aggressive way to deal with this: periodic LOOCV
- *“...input signals that provide little incremental modeling benefit...can make an ML system **unnecessarily vulnerable to change**, sometimes catastrophically so, even though they could be removed with no detriment.”*

IV. Feedback Loops

- **ML systems often end up influencing their own behavior if they update over time**
 - **Analysis debt:** it is difficult to predict the behavior of a given model before it is released
- **Direct feedback loops:** a model may directly influence the selection of its own future training data
 - Can mitigate with bandits, but they don't scale well to the size of real-world action spaces
 - Can mitigate with randomization, i.e., isolate some of the data/traffic from model influence
- **Hidden feedback loops:** two systems influence each other indirectly through the world
 - E.g., two stock-market prediction models from two different investment companies.
Improvements (or, more scarily, bugs) in one may influence the bidding and buying behavior of the other

V. ML Antipatterns

- **Glue Code**

- Generalized packages like sklearn, keras, etc are popular so 95% of the code you write is "glue code" that transforms your inputs/outputs to fit the package.
- This can inhibit improvements to the algorithm itself like tweaking the cost function. Also you spend so much time writing glue code you might be better off implementing it yourself.

- **Pipeline Jungles**

- New features get added as dependencies in ways that were not originally planned for and you get a chaotic mess.
- They suggest refactoring from start and combining research with engineering skill sets.

- **Dead experimental codepaths**

- It's tempting to develop off experimental branches where you can tweak other parts of the system, and apparently it's even more tempting to keep branching off that into further experiments without merging to master?

V. ML Antipatterns

- **Abstraction Debt**

- Authors say there aren't as many powerful abstractions for ML as for CS (example: databases).
- I think there's been good work done in the industry but this is something I think we need to talk about @ Stash. How do we define a model, retraining, eval, etc. can we find a standard format to use?
- **"Lack of abstractions makes it easy to blur lines between components"** -> yes, and also causes tech debt because it takes longer to understand other people's code, you spend more time writing boiler-plate, etc. I agree with this point the most.

- **"Smells"**: they identify 3 warning signs that may indicate a poor ML system

- Plain-Old-Data Type Smell: **"The rich information used and produced by ML systems is all too often encoded with plain data types like raw floats and integers. In a robust system, a model parameter should know if it is a log-odds multiplier or a decision threshold, and a prediction should know various pieces of information about the model that produced it and how it should be consumed."** -> I disagree with this, let's discuss
- Multiple language smell: using multiple langs increases tech debt
- Prototype smell: if you can't move your code from edge to prod with ease then your prod env is too complicated and could use some abstractions or refactoring.

VI. Configuration Debt

- **The Problem**

- Lines of configuration code grow quick, it may be even faster than normal code
- Each new line of config code is a potential failure.
- Engineers tend to downplay the testing and documenting of config code Intertwined configurations can lead to dependency issues & mangled code that is hard to understand

- **Rules of good config systems:**

- It should be easy to specify a configuration as a small change from a previous configuration. -> how? Comments, clear variable names, a lot of examples, logical design
- It should be hard to make manual errors, omissions, or oversights. -> how? Unit tests!!! Code review/ownership schema
- Its should be easy to see, visually, the difference in configuration between two models. -> interesting, I think this leads back to my earlier point about abstracting models @ stash It should be easy to automatically assert and verify basic facts about the configuration: number of features used, transitive closure of data dependencies, etc. It should be possible to detect unused or redundant settings.
- Configurations should undergo a full code review and be checked into a repository.

VII. Dealing With Changes In The External World

- **Fixed thresholds in dynamic systems**

- Often we pick a threshold based on training data and business logic. Ie a FP-FN threshold.
- Often we pick this threshold once and dont adjust it, but the underlying data changes when we deploy.
- Dynamic/periodic retraining/recomputation should eliminate this.

- **Monitoring and testing:** unit tests and integration tests do not suffice when your models rely on real world data. Here's what you should be monitoring..

- **Prediction bias:** distribution of predicted labels should match distribution of actual labels. This is a smoke test, you can have a bad model where this is true but you can't have a good model where its not. Will alert you to sharp changes in underlying data & you should retrain the model.
- **Action limits:** systems that are used to take actions in the real world (bidding, marketing messages) should have action limits that prevent the system issuing too many & send out alerts
- **Upstream producers** : if any of your upstream data sources fails in prod, so does your model. Be aware of this and the real word implications

VIII. Other Areas of ML-related Debt

- **Data Testing Debt:** If data replaces code in ML systems, testing of input data is critical (monitor changes in input distribution).
- **Reproducibility Debt:** strict reproducibility in real-world is difficult.
- **Process Management Debt:** Mature systems may have dozens of models running simultaneously, developing tools to aid recovery from production incidents is critical.
- **Cultural Debt:** It's important to create team cultures that reward deletion of features, reduction of complexity, improvements in reproducibility, stability, and monitoring to the same degree that improvements in accuracy are valued (everyone on the same page)

IX. Conclusions: Measuring Debt and Paying it Off

- **Measuring**
 - How easily can an entirely new algorithmic approach be tested at full scale?
 - What is the transitive closure of all data dependencies?
 - How precisely can the impact of a new change to the system be measured?
 - Does improving one model or signal degrade others?
 - How quickly can new members of the team be brought up to speed?
- *Research solutions that provide a tiny accuracy benefit at the cost of massive increases in system complexity are rarely wise practice*
- *Paying down ML-related technical debt requires a specific commitment, which can often only be achieved by a shift in team culture. Recognizing, prioritizing, and rewarding this effort is important for the long term health of successful ML teams.*