

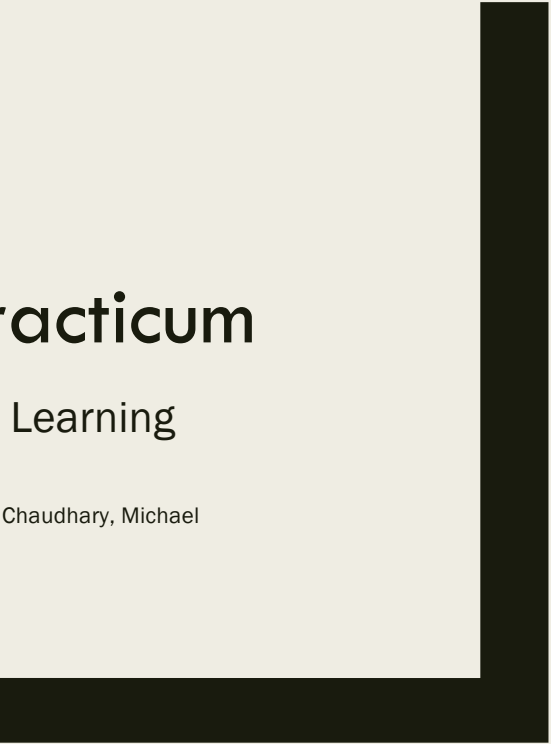


CSCI 8360: Data Science Practicum

Lecture 4: “Hidden Technical Debt in Machine Learning Systems”

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, Dan Dennison

Dr. Shannon Quinn



What is “technical debt”?

- Coined by Ward Cunningham in 1992
 - *Refers to long-term costs incurred by moving quickly in software engineering*
- Debt metaphor
 - *Not necessarily a bad thing, but **always** needs to be serviced*
- Goal: **NOT** to add new functionality
 - *Enable future improvements, reduce errors, improve maintainability*

What is “technical debt”?

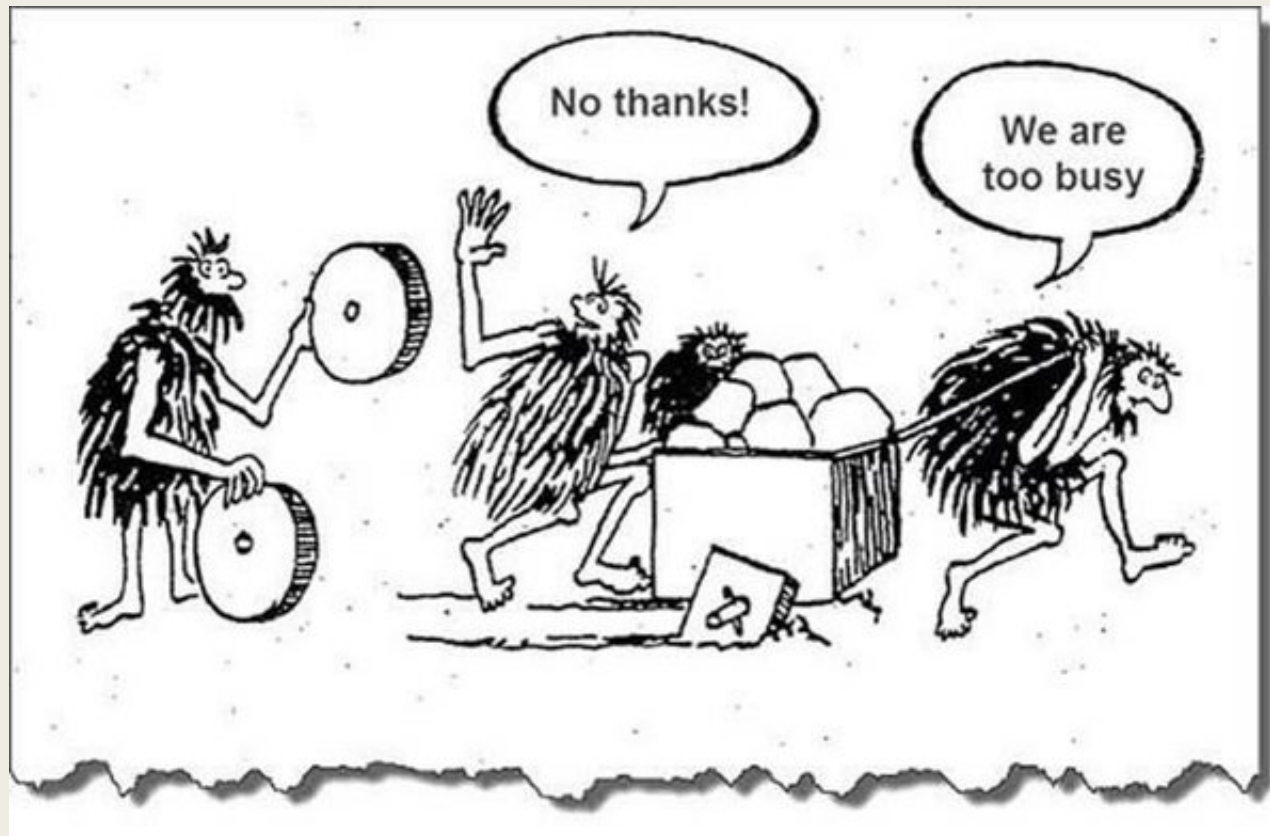
Technical debt

From Wikipedia, the free encyclopedia

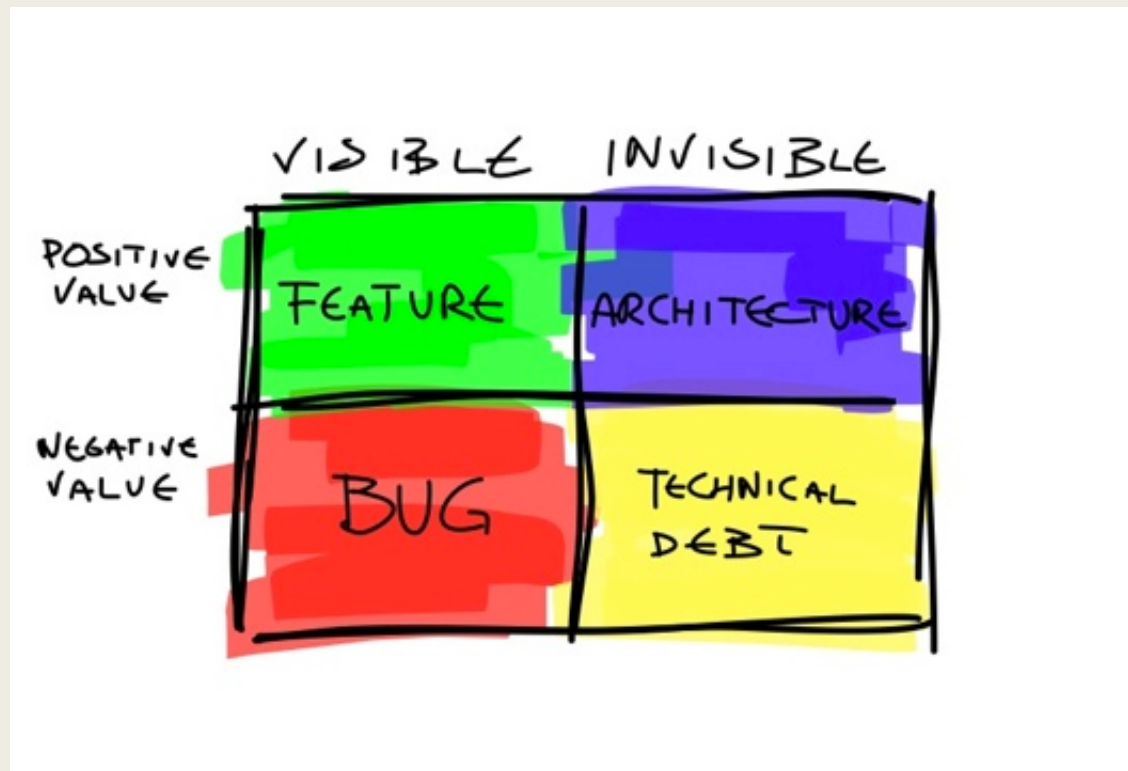
Technical debt (also known as **design debt**^[1] or **code debt**) is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution^[2]".

Technical debt can be compared to monetary [debt](#).^[3] If technical debt is not repaid, it can accumulate 'interest', making it harder to implement changes later on. Unaddressed technical debt increases [software entropy](#). Technical debt is not necessarily a bad thing, and sometimes (e.g., as a proof-of-concept) technical debt is required to move projects forward. On the other hand, some experts claim that the "technical debt" metaphor tends to minimize the impact, which results in insufficient prioritization of the necessary work to correct it.^{[4][5]}

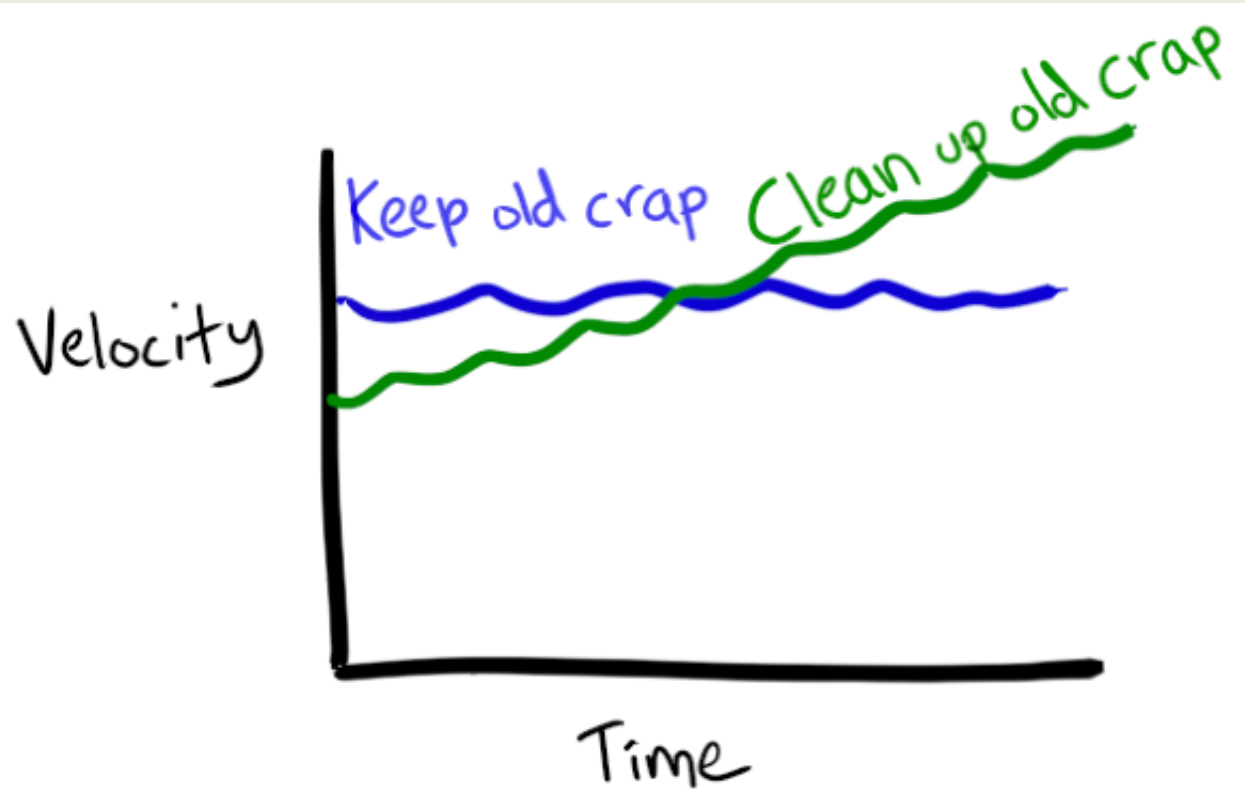
What is “technical debt”?



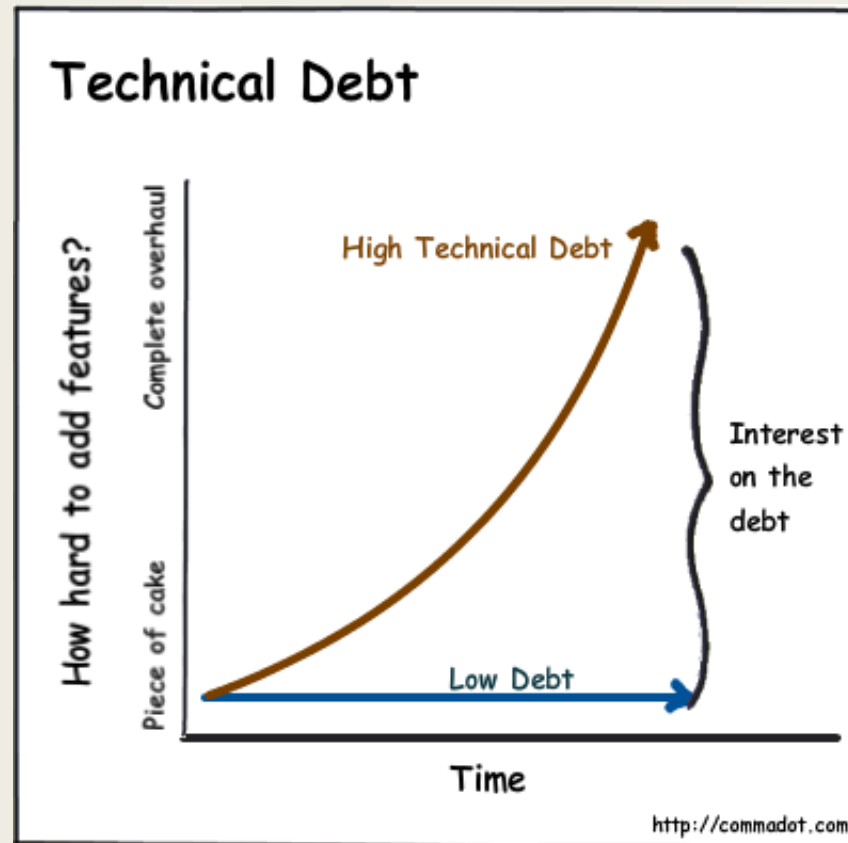
What is “technical debt”?



What is “technical debt”?



What is “technical debt”?



Causes of technical debt

Technical Debt and Machine Learning

- All the maintenance problems of “traditional” code
 - *Plus an additional set of ML-specific concerns*
- Debt can exist at system level, instead of [strictly] code level
 - *Data influences ML system behavior!*
 - *“Traditional” abstractions and boundaries can be corrupted*
- “Traditional” methods for paying down code-level debt are not sufficient to address ML-specific issues at system level

1: Model Complexity

- Traditional software engineering: strong abstraction boundaries, encapsulation, and modular design
- Machine learning: desired behavior relies *specifically* on external data
 - *The real world does not fit into tidy abstraction rules*



1: Model Complexity

■ Entanglement

- *ML systems mix signals*
- *If we change distribution of one input feature, weights of other $d - 1$ features may change as well*
- *“Changing Anything Changes Everything”*

■ Correction Cascades

- *We have model m_a for problem A , but need a solution for slightly different A'*
- *Tweak m_a to m'_a ...then need to solve A'' , and so on*

■ Undeclared Consumers

- *Output of your model m_a may be input to some downstream system*
- *Changes in your model m_a will drastically affect performance of downstream consumers*

2: Data Dependencies

- Traditional software engineering identifies “dependency debt” as a key contributor to overall technical debt
- Data dependencies in ML systems carry similar debt-building capacity, but with the added joy of being more difficult to detect

2: Data Dependencies

■ Unstable Data Dependencies

- *Consumed input changes over time*
- *Input signal comes from another ML system that updates itself*
- *Engineering ownership of input signal is distinct from ML system consuming it*

■ Underutilized Data Dependencies

- *Input signals that provide little modeling benefit*
- *Legacy features, bundled features, correlated features, ϵ -features*

3: Feedback Loops

- A key feature of ML systems is influencing its own behavior
- Analysis debt
 - *Difficult to predict behavior of a given model before release**

* Tay?

3: Feedback Loops

■ Direct Feedback Loops

- *Model directly influences selection of its own future training data*
- *Supervised algorithms*

■ Hidden Feedback Loops

- *Two systems indirectly influence each other through the world*
- *Related but distinct recommendation systems—improving one leads to changes in the other*

4: ML System Anti-patterns

- How much code in a machine learning system is, well, *machine learning*?
- The non-ML code is “plumbing”—the majority, but nonetheless typically an afterthought.

4: ML System Anti-patterns

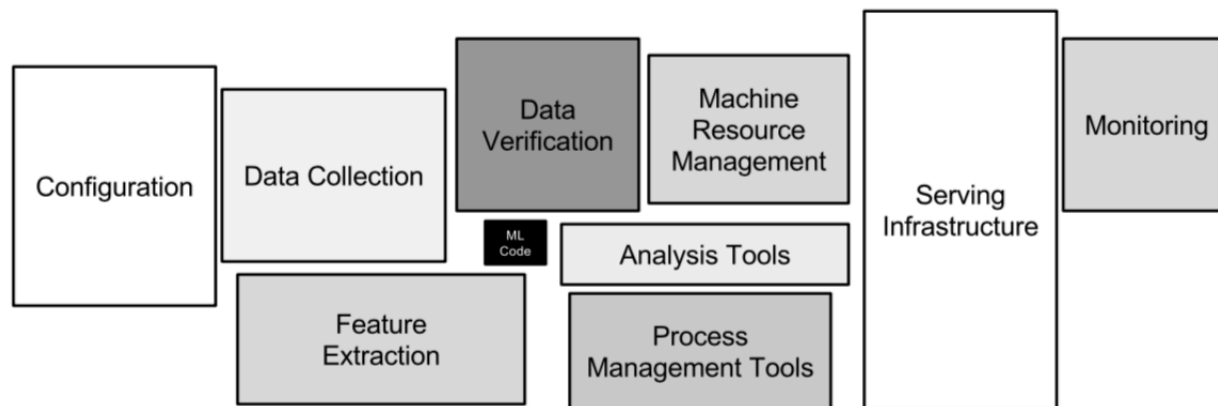


Figure 1: Only a small fraction of real-world ML systems is composed of the ML code, as shown by the small black box in the middle. The required surrounding infrastructure is vast and complex.

5: Configuration Debt

- Large systems have wide range of configurable options
 - *Features used*
 - *Data selection*
 - *Algorithm hyperparameters*
 - *Verification methods*
 - *Pre- and post-processing routines*
- May reach the point of **# of lines of configuration >>> # of lines of code**

5: Configuration Debt

- It should be easy to specify a configuration as a small change from a previous configuration.
- It should be hard to make manual errors, omissions, or oversights.
- It should be easy to see, visually, the difference in configuration between two models.
- 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.

6: Changes in the External World

- Related to #2 "Data Dependencies" and #3 "Feedback Loops"
- Fixed Thresholds in Dynamic Systems
 - *What $p(x)$ will be used to separate "spam" from "not spam"?*
- Monitoring and Testing
 - *What to monitor?*
 - *Prediction Bias (distribution of predicted labels)*
 - *Action Limits (automated alerts)*
- Upstream Producers
 - *Any upstream data producers should also be thoroughly and frequently tested*

7: Other ML-related Debt

- **Data Testing Debt**

- *If data == code in ML systems, and code should be tested, then some data should be tested as well to monitor for changes in input distributions*

- **Reproducibility Debt (Open Science!)**

- *If frameworks are re-run in identical configurations, should produce identical results*

- **Process Management Debt**

- *Mature systems may have dozens or even hundreds of simultaneous ML models*

- **Cultural Debt**

- *Create a healthy project team culture with a breadth of expertise that rewards good engineering practices*

Conclusions

- Measuring technical debt
 - *No clear metric*
- Simply “moving fast” is not evidence of low debt or good practices
 - *Moving quickly often **introduces** technical debt!*
- Useful questions for consideration:
 1. *How easily can an entirely new algorithmic approach be tested at full scale?*
 2. *How precisely can the impact of a new change to the system be measured?*
 3. *Does improving one model or signal degrade others?*
 4. *How quickly can new members of the team be brought up to speed?*

Questions?

