Machine Learning: The High-Interest Credit Card of Technical Debt

D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov,
Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young
{dsculley,gholt,dgg,edavydov}@google.com
{toddphillips,ebner,vchaudhary,mwyoung}@google.com
Google, Inc

Abstract

Machine learning offers a fantastically powerful toolkit for building complex systems quickly. This paper argues that it is dangerous to think of these quick wins as coming for free. Using the framework of *technical debt*, we note that it is remarkably easy to incur massive ongoing maintenance costs at the system level when applying machine learning. The goal of this paper is highlight several machine learning specific risk factors and design patterns to be avoided or refactored where possible. These include boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, changes in the external world, and a variety of system-level anti-patterns.

1 Machine Learning and Complex Systems

Real world software engineers are often faced with the challenge of moving quickly to ship new products or services, which can lead to a dilemma between speed of execution and quality of engineering. The concept of *technical debt* was first introduced by Ward Cunningham in 1992 as a way to help quantify the cost of such decisions. Like incurring fiscal debt, there are often sound strategic reasons to take on technical debt. Not all debt is necessarily bad, but technical debt does tend to compound. Deferring the work to pay it off results in increasing costs, system brittleness, and reduced rates of innovation.

Traditional methods of paying off technical debt include refactoring, increasing coverage of unit tests, deleting dead code, reducing dependencies, tightening APIs, and improving documentation [4]. The goal of these activities is *not* to add new functionality, but to make it easier to add future improvements, be cheaper to maintain, and reduce the likelihood of bugs.

One of the basic arguments in this paper is that machine learning packages have all the basic code complexity issues as normal code, but also have a larger system-level complexity that can create hidden debt. Thus, refactoring these libraries, adding better unit tests, and associated activity is time well spent but does not necessarily address debt at a systems level.

In this paper, we focus on the system-level interaction between machine learning code and larger systems as an area where hidden technical debt may rapidly accumulate. At a system-level, a machine learning model may subtly erode abstraction boundaries. It may be tempting to re-use input signals in ways that create unintended tight coupling of otherwise disjoint systems. Machine learning packages may often be treated as black boxes, resulting in large masses of "glue code" or calibration layers that can lock in assumptions. Changes in the external world may make models or input signals change behavior in unintended ways, ratcheting up maintenance cost and the burden of any debt. Even monitoring that the system as a whole is operating as intended may be difficult without careful design.

Indeed, a remarkable portion of real-world "machine learning" work is devoted to tackling issues of this form. Paying down technical debt may initially appear less glamorous than research results usually reported in academic ML conferences. But it is critical for long-term system health and enables algorithmic advances and other cutting-edge improvements.

2 Complex Models Erode Boundaries

Traditional software engineering practice has shown that strong abstraction boundaries using encapsulation and modular design help create maintainable code in which it is easy to make isolated changes and improvements. Strict abstraction boundaries help express the invariants and logical consistency of the information inputs and outputs from an given component [4].

Unfortunately, it is difficult to enforce strict abstraction boundaries for machine learning systems by requiring these systems to adhere to specific intended behavior. Indeed, arguably the most important reason for using a machine learning system is precisely that *the desired behavior cannot be effectively implemented in software logic without dependency on external data*. There is little way to separate abstract behavioral invariants from quirks of data. The resulting erosion of boundaries can cause significant increases in technical debt. In this section we look at several issues of this form.

2.1 Entanglement

From a high level perspective, a machine learning package is a tool for mixing data sources together. That is, machine learning models are machines for creating entanglement and making the isolation of improvements effectively impossible.

To make this concrete, imagine we have a system that uses features $\mathbf{x}_1, ...\mathbf{x}_n$ in a model. If we change the input distribution of values in $\mathbf{x}_1$, the importance, weights, or use of the remaining $n-1$ features may all change—this is true whether the model is retrained fully in a batch style or allowed to adapt in an online fashion. Adding a new feature $\mathbf{x}_{n+1}$ can cause similar changes, as can removing any feature $\mathbf{x}_j$. No inputs are ever really independent. We refer to this here as the CACE principle: Changing Anything Changes Everything.

The net result of such changes is that prediction behavior may alter, either subtly or dramatically, on various slices of the distribution. The same principle applies to hyper-parameters. Changes in regularization strength, learning settings, sampling methods in training, convergence thresholds, and essentially every other possible tweak can have similarly wide ranging effects.

One possible mitigation strategy is to isolate models and serve ensembles. This approach is useful in situations such as [8], in which sub-problems decompose naturally, or in which the cost of maintaining separate models is outweighed by the benefits of enforced modularity. However, in many large-scale settings such a strategy may prove unscalable. And within a given model, the issues of entanglement may still be present.

A second possible mitigation strategy is to develop methods of gaining deep insights into the behavior of model predictions. One such method was proposed in [6], in which a high-dimensional visualization tool was used to allow researchers to quickly see effects across many dimensions and slicings. Metrics that operate on a slice-by-slice basis may also be extremely useful.

A third possibility is to attempt to use more sophisticated regularization methods to enforce that any changes in prediction performance carry a cost in the objective function used in training [5]. Like any other regularization approach, this kind of approach can be useful but is far from a guarantee and may add more debt via increased system complexity than is reduced via decreased entanglement.

The above mitigation strategies may help, but this issue of entanglement is in some sense innate to machine learning, regardless of the particular learning algorithm being used. In practice, this all too often means that shipping the first version of a machine learning system is easy, but that making subsequent improvements is unexpectedly difficult. This consideration should be weighed carefully against deadline pressures for version 1.0 of any ML system.

2.2 Hidden Feedback Loops

Another worry for real-world systems lies in hidden feedback loops. Systems that learn from world behavior are clearly intended to be part of a feedback loop. For example, a system for predicting the click through rate (CTR) of news headlines on a website likely relies on user clicks as training labels, which in turn depend on previous predictions from the model. This leads to issues in analyzing system performance, but these are the obvious kinds of statistical challenges that machine learning researchers may find natural to investigate [2].

As an example of a *hidden* loop, now imagine that one of the input features used in this CTR model is a feature $\mathbf{x}_{week}$ that reports how many news headlines the given user has clicked on in the past week. If the CTR model is improved, it is likely that all users are given better recommendations and many users will click on more headlines. However, the result of this effect may not fully surface for at least a week, as the $\mathbf{x}_{week}$ feature adjusts. Furthermore, if the model is updated on the new data, either in batch mode at a later time or in streaming fashion with online updates, the model may later adjust its opinion of the $\mathbf{x}_{week}$ feature in response. In such a setting, the system will slowly change behavior, potentially over a time scale much longer than a week. Gradual changes not visible in quick experiments make analyzing the effect of proposed changes extremely difficult, and add cost to even simple improvements.

We recommend looking carefully for hidden feedback loops and removing them whenever feasible.

2.3 Undeclared Consumers

Oftentimes, a prediction from a machine learning model A is made accessible to a wide variety of systems, either at runtime or by writing to logs that may later be consumed by other systems. In more classical software engineering, these issues are referred to as visibility debt [7]. Without access controls, it is possible for some of these consumers to be *undeclared* consumers, consuming the output of a given prediction model as an input to another component of the system. Undeclared consumers are expensive at best and dangerous at worst.

The expense of undeclared consumers is drawn from the sudden tight coupling of model A to other parts of the stack. Changes to A will very likely impact these other parts, sometimes in ways that are unintended, poorly understood, or detrimental. In practice, this has the effect of making it difficult and expensive to make any changes to A at all.

The danger of undeclared consumers is that they may introduce additional hidden feedback loops. Imagine in our news headline CTR prediction system that there is another component of the system in charge of “intelligently” determining the size of the font used for the headline. If this font-size module starts consuming CTR as an input signal, and font-size has an effect on user propensity to click, then the inclusion of CTR in font-size adds a new hidden feedback loop. It’s easy to imagine a case where such a system would gradually and endlessly increase the size of all headlines.

Undeclared consumers may be difficult to detect unless the system is specifically designed to guard against this case. In the absence of barriers, engineers may naturally grab for the most convenient signal, especially when there are deadline pressures.

3 Data Dependencies Cost More than Code Dependencies

In [7], *dependency debt* is noted as a key contributor to code complexity and technical debt in classical software engineering settings. We argue here that *data dependencies* in machine learning systems carry a similar capacity for building debt. Furthermore, while code dependencies can be relatively easy to identify via static analysis, linkage graphs, and the like, it is far less common that data dependencies have similar analysis tools. Thus, it can be inappropriately easy to build large data-dependency chains that can be difficult to untangle.

3.1 Unstable Data Dependencies

To move quickly, it is often convenient to consume signals as input features that are produced by other systems. However, some input signals are *unstable*, meaning that they qualitatively change behavior over time. This can happen implicitly, when the input signal comes from another machine learning model itself that updates over time, or a data-dependent lookup table, such as for computing TF/IDF scores or semantic mappings. It can also happen explicitly, when the engineering ownership of the input signal is separate from the engineering ownership of the model that consumes it. In such cases, changes and improvements to the input signal may be regularly rolled out, without regard for how the machine learning system may be affected. As noted above in the CACE principle, "improvements" to input signals may have arbitrary, sometimes deleterious, effects that are costly to diagnose and address.

One common mitigation strategy for unstable data dependencies is to create a *versioned copy* of a given signal. For example, rather than allowing a semantic mapping of words to topic clusters to change over time, it might be reasonable to create a frozen version of this mapping and use it until such a time as an updated version has been fully vetted. Versioning carries its own costs, however, such as potential staleness. And the requirement to maintain multiple versions of the same signal over time is a contributor to technical debt in its own right.

3.2 Underutilized Data Dependencies

In code, underutilized dependencies are packages that are mostly unneeded [7]. Similarly, underutilized data dependencies include input features or signals that provide little incremental value in terms of accuracy. Underutilized dependencies are costly, since they make the system unnecessarily vulnerable to changes.

Underutilized dependencies can creep into a machine learning model in several ways.

Legacy Features. The most common is that a feature F is included in a model early in its development. As time goes on, other features are added that make F mostly or entirely redundant, but this is not detected.

Bundled Features. Sometimes, a group of features is evaluated and found to be beneficial. Because of deadline pressures or similar effects, all the features in the bundle are added to the model together. This form of process can hide features that add little or no value.

ϵ-Features. As machine learning researchers, it is satisfying to improve model accuracy. It can be tempting to add a new feature to a model that improves accuracy, even when the accuracy gain is very small or when the complexity overhead might be high.

In all these cases, features could be removed from the model with little or no loss in accuracy. But because they are still present, the model will likely assign them some weight, and the system is therefore vulnerable, sometimes catastrophically so, to changes in these unnecessary features.

As an example, suppose that after a team merger, to ease the transition from an old product numbering scheme to new product numbers, both schemes are left in the system as features. New products get only a new number, but old products may have both. The machine learning algorithm knows of no reason to reduce its reliance on the old numbers. A year later, someone acting with good intent cleans up the code that stops populating the database with the old numbers. This change goes undetected by regression tests because no one else is using them any more. This will not be a good day for the maintainers of the machine learning system.

A common mitigation strategy for under-utilized dependencies is to regularly evaluate the effect of removing individual features from a given model and act on this information whenever possible. More broadly, it may be important to build cultural awareness about the long-term benefit of underutilized dependency cleanup.

3.3 Static Analysis of Data Dependencies

One of the key issues in data dependency debt is the difficulty of performing static analysis. While compilers and build systems typically provide such functionality for code, data dependencies may require additional tooling to track. Without this, it can be difficult to manually track the use of data in a system. On teams with many engineers, or if there are multiple interacting teams, not everyone knows the status of every single feature, and it can be difficult for any individual human to know every last place where the feature was used. For example, suppose that the version of a dictionary must be changed; in a large company, it may not be easy even to find all the consumers of the dictionary. Or suppose that for efficiency a particular signal will no longer be computed; are all former consumers of the signal done with it? Even if there are no references to it in the current version of the codebase, are there still production instances with older binaries that use it? Making changes safely can be difficult without automatic tooling.

A remarkably useful automated feature management tool was described in [6], which enables data sources and features to be annotated. Automated checks can then be run to ensure that all dependencies have the appropriate annotations, and dependency trees can be fully resolved. Since its adoption, this approach has regularly allowed a team at Google to safely delete thousands of lines of feature-related code per quarter, and has made verification of versions and other issues automatic. The system has on many occasions prevented accidental use of deprecated or broken features in new models.

3.4 Correction Cascades

There are often situations in which model a for problem A exists, but a solution for a slightly different problem A' is required. In this case, it can be tempting to learn a model $a'(a)$ that takes a as input and learns a small correction. This can appear to be a fast, low-cost win, as the correction model is likely very small and can often be done by a completely independent team. It is easy and quick to create a first version.

However, this correction model has created a system dependency on a, making it significantly more expensive to analyze improvements to that model in the future. Things get even worse if correction models are cascaded, with a model for problem A'' learned on top of a', and so on. This can easily happen for closely related problems, such as calibrating outputs to slightly different test distributions. It is not at all unlikely that a correction cascade will create a situation where improving the accuracy of a actually leads to system-level detriments. Additionally, such systems may create *deadlock*, where the coupled ML system is in a poor local optimum, and no component model may be individually improved. At this point, the independent development that was initially attractive now becomes a large barrier to progress.

A mitigation strategy is to augment a to learn the corrections directly within the same model by adding features that help the model distinguish among the various use-cases. At test time, the model may be queried with the appropriate features for the appropriate test distributions. This is not a free solution—the solutions for the various related problems remain coupled via CACE, but it may be easier to make updates and evaluate their impact.

4 System-level Spaghetti

It is unfortunately common for systems that incorporate machine learning methods to end up with high-debt design patterns. In this section, we examine several system-design *anti-patterns* [3] that can surface in machine learning systems and which should be avoided or refactored where possible.

4.1 Glue Code

Machine learning researchers tend to develop general purpose solutions as self-contained packages. A wide variety of these are available as open-source packages at places like `mloss.org`, or from in-house code, proprietary packages, and cloud-based platforms. Using self-contained solutions often results in a *glue code* system design pattern, in which a massive amount of supporting code is written to get data into and out of general-purpose packages.

This glue code design pattern can be costly in the long term, as it tends to freeze a system to the peculiarities of a specific package. General purpose solutions often have different design goals: they seek to provide one learning system to solve many problems, but many practical software systems are highly engineered to apply to one large-scale problem, for which many experimental solutions are sought. While generic systems might make it possible to interchange optimization algorithms, it is quite often refactoring of the *construction* of the problem space which yields the most benefit to mature systems. The glue code pattern implicitly embeds this construction in supporting code instead of in principally designed components. As a result, the glue code pattern often makes experimentation with other machine learning approaches prohibitively expensive, resulting in an ongoing tax on innovation.

Glue code can be reduced by choosing to re-implement specific algorithms within the broader system architecture. At first, this may seem like a high cost to pay—re-implementing a machine learning package in C++ or Java that is already available in **R** or `matlab`, for example, may appear to be a waste of effort. But the resulting system may require dramatically less glue code to integrate in the overall system, be easier to test, be easier to maintain, and be better designed to allow alternate approaches to be plugged in and empirically tested. Problem-specific machine learning code can also be tweaked with problem-specific knowledge that is hard to support in general packages.

It may be surprising to the academic community to know that only a tiny fraction of the code in many machine learning systems is actually doing "machine learning". When we recognize that a mature system might end up being (at most) 5% machine learning code and (at least) 95% glue code, reimplementation rather than reuse of a clumsy API looks like a much better strategy.

4.2 Pipeline Jungles

As a special case of glue code, *pipeline jungles* often appear in data preparation. These can evolve organically, as new signals are identified and new information sources added. Without care, the resulting system for preparing data in an ML-friendly format may become a jungle of scrapes, joins, and sampling steps, often with intermediate files output. Managing these pipelines, detecting errors and recovering from failures are all difficult and costly [1]. Testing such pipelines often requires expensive end-to-end integration tests. All of this adds to technical debt of a system and makes further innovation more costly.

Pipeline jungles can only be avoided by thinking holistically about data collection and feature extraction. The clean-slate approach of scrapping a pipeline jungle and redesigning from the ground up is indeed a major investment of engineering effort, but one that can dramatically reduce ongoing costs and speed further innovation.

It's worth noting that glue code and pipeline jungles are symptomatic of integration issues that may have a root cause in overly separated "research" and "engineering" roles. When machine learning packages are developed in an ivory-tower setting, the resulting packages may appear to be more like black boxes to the teams that actually employ them in practice. At Google, a hybrid research approach where engineers and researchers are embedded together on the same teams (and indeed, are often the same people) has helped reduce this source of friction significantly [10]. But even when a fully integrated team structure is not possible, it can be advantageous to have close, active collaborations.

4.3 Dead Experimental Codepaths

A common reaction to the hardening of glue code or pipeline jungles is that it becomes more and more tempting to perform experiments with alternative algorithms or tweaks by implementing these experimental codepaths as conditional branches within the main production code. For any individual change, the cost of experimenting in this manner is relatively low—none of the surrounding infrastructure needs to be reworked. However, over time, these accumulated codepaths can create a growing debt. Maintaining backward compatibility with experimental codepaths is a burden for making more substantive changes. Furthermore, obsolete experimental codepaths can interact with each other in unpredictable ways, and tracking which combinations are incompatible quickly results in an exponential blowup in system complexity. A famous example of the dangers here was Knight Capital's system losing $465 million in 45 minutes apparently because of unexpected behavior from obsolete experimental codepaths [9].

As with the case of *dead flags* in traditional software [7], it is often beneficial to periodically re-examine each experimental branch to see what can be ripped out. Very often only a small subset of the possible branches is actually used; many others may have been tested once and abandoned.

Dead experimental codepaths are a symptom of a more fundamental issue: in a healthy machine learning system, experimental code should be well isolated, not leaving tendrils in multiple modules. This may require rethinking code APIs. In our experience, the kinds of things we want to experiment with vary over time; a redesign and a rewrite of some pieces may be needed periodically in order to move forward efficiently.

As a real-world anecdote, in a recent cleanup effort of one important machine learning system at Google, it was found possible to rip out tens of thousands of lines of unused experimental code-paths. A follow-on rewrite with a tighter API allowed experimentation with new algorithms to be performed with dramatically reduced effort and production risk and minimal incremental system complexity.

4.4 Configuration Debt

Another potentially surprising area where debt can accumulate is in the configuration of machine learning systems. Any large system has a wide range of configurable options, including which features are used, how data is selected, a wide variety of algorithm-specific learning settings, potential pre- or post-processing, verification methods, etc.

Many engineers do a great job of thinking hard about abstractions and unit tests in production code, but may treat configuration (and extension of configuration) as an afterthought. Indeed, verification or testing of configurations may not even be seen as important. Configuration by its very nature tends to be the place where real-world messiness intrudes on beautiful algorithms.

Consider the following examples. Feature A was incorrectly logged from 9/14 to 9/17. Feature B is not available on data before 10/7. The code used to compute feature C has to change for data before and after 11/1 because of changes to the logging format. Feature D is not available in production, so a substitute features D' and D'' must be used when querying the model in a live setting. If feature Z is used, then jobs for training must be given extra memory due to lookup tables or they will train inefficiently. Feature Q precludes the use of feature R because of latency constraints. All this messiness makes configuration hard to modify correctly, and hard to reason about. However, mistakes in configuration can be costly, leading to serious loss of time, waste of computing resources, or production issues.

Also, in a mature system which is being actively developed, the number of lines of configuration can far exceed the number of lines of the code that actually does machine learning. Each line has a potential for mistakes, and configurations are by their nature ephemeral and less well tested.

Assertions about configuration invariants can be critical to prevent mistakes, but careful thought is needed about what kind of assertions will be useful. Another useful tool is the ability to present visual side-by-side differences (diffs) of two configurations. Because configurations are often copy-and-pasted with small modifications, such diffs highlight important changes. And clearly, configurations should be treated with the same level of seriousness as code changes, and be carefully code reviewed by peers.

5 Dealing with Changes in the External World

One of the things that makes machine learning systems so fascinating is that they often interact directly with the external world. Experience has shown that the external world is rarely stable. Indeed, the changing nature of the world is one of the sources of technical debt in machine learning systems.

5.1 Fixed Thresholds in Dynamic Systems

It is often necessary to pick a *decision threshold* for a given model to perform some action: to predict true or false, to mark an email as spam or not spam, to show or not show a given ad. One classic approach in machine learning is to choose a threshold from a set of possible thresholds, in order to get good tradeoffs on certain metrics, such as precision and recall. However, such thresholds are often manually set. Thus if a model updates on new data, the old manually set threshold may be invalid. Manually updating many thresholds across many models is time-consuming and brittle.

A useful mitigation strategy for this kind of problem appears in [8], in which thresholds are learned via simple evaluation on heldout validation data.

5.2 When Correlations No Longer Correlate

Machine learning systems often have a difficult time distinguishing the impact of correlated features. This may not seem like a major problem: if two features are always correlated, but only one is truly causal, it may still seem okay to ascribe credit to both and rely on their observed co-occurrence.

However, if the world suddenly stops making these features co-occur, prediction behavior may change significantly. The full range of ML strategies for teasing apart correlation effects is beyond our scope; some excellent suggestions and references are given in [2]. For the purpose of this paper, we note that non-causal correlations are another source of hidden debt.

5.3 Monitoring and Testing

Unit testing of individual components and end-to-end tests of running systems are valuable, but in the face of a changing world such tests are not sufficient to provide evidence that a system is working as intended. Live monitoring of system behavior in real time is critical.

The key question is: what to monitor? It can be difficult to establish useful invariants, given that the purpose of machine learning systems is to adapt over time. We offer two reasonable starting points.

Prediction Bias. In a system that is working as intended, it should usually be the case that the distribution of predicted labels is equal to the distribution of observed labels. This is by no means a comprehensive test, as it can be met by a null model that simply predicts average values of label occurrences without regard to the input features. However, it is a surprisingly useful diagnostic, and changes in metrics such as this are often indicative of an issue that requires attention. For example, this method can help to detect cases in which the world behavior suddenly changes, making training distributions drawn from historical data no longer reflective of current reality. Slicing prediction bias by various dimensions isolate issues quickly, and can also be used for automated alerting.

Action Limits. In systems that are used to take actions in the real world, it can be useful to set and enforce action limits as a sanity check. These limits should be broad enough not to trigger spuriously. If the system hits a limit for a given action, automated alerts should fire and trigger manual intervention or investigation.

6 Conclusions: Paying it Off

This paper has highlighted a number of areas where machine learning systems can create technical debt, sometimes in surprising ways. This is not to say that machine learning is bad, or even that technical debt is something to be avoided at all costs. It may be reasonable to take on moderate technical debt for the benefit of moving quickly in the short term, but this must be recognized and accounted for lest it quickly grow unmanageable.

Perhaps the most important insight to be gained is that technical debt is an issue that *both* engineers and researchers need to be aware of. Research solutions that provide a tiny accuracy benefit at the cost of massive increases in system complexity are rarely wise practice. Even the addition of one or two seemingly innocuous data dependencies can slow further progress.

Paying down technical debt is not always as exciting as proving a new theorem, but it is a critical part of consistently strong innovation. And developing holistic, elegant solutions for complex machine learning systems is deeply rewarding work.

Acknowledgments

This paper owes much to the important lessons learned day to day in a culture that values both innovative ML research and strong engineering practice. Many colleagues have helped shape our thoughts here, and the benefit of accumulated folk wisdom cannot be overstated. We would like to specifically recognize the following: Luis Cobo, Sharat Chikkerur, Jean-Francois Crespo, Jeff Dean, Dan Dennison, Philip Henderson, Arnar Mar Hrafnkelsson, Ankur Jain, Joe Kovac, Jeremy Kubica, H. Brendan McMahan, Satyaki Mahalanabis, Lan Nie, Michael Pohl, Abdul Salem, Sajid Siddiqi, Ricky Shan, Alan Skelly, Cory Williams, and Andrew Young.

References

[1] R. Ananthanarayanan, V. Basker, S. Das, A. Gupta, H. Jiang, T. Qiu, A. Reznichenko, D. Ryabkov, M. Singh, and S. Venkataraman. Photon: Fault-tolerant and scalable joining of continuous data streams. In *SIGMOD '13: Proceedings of the 2013 international conference on Management of data*, pages 577–588, New York, NY, USA, 2013.

[2] L. Bottou, J. Peters, J. Quiñonero Candela, D. X. Charles, D. M. Chickering, E. Portugaly, D. Ray, P. Simard, and E. Snelson. Counterfactual reasoning and learning systems: The example of computational advertising. *Journal of Machine Learning Research*, 14(Nov), 2013.

[3] W. J. Brown, H. W. McCormick, T. J. Mowbray, and R. C. Malveau. Antipatterns: refactoring software, architectures, and projects in crisis. 1998.

[4] M. Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[5] A. Lavoie, M. E. Otey, N. Ratliff, and D. Sculley. History dependent domain adaptation. In *Domain Adaptation Workshop at NIPS '11*, 2011.

[6] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, D. Golovin, S. Chikkerur, D. Liu, M. Wattenberg, A. M. Hrafnkelsson, T. Boulos, and J. Kubica. Ad click prediction: a view from the trenches. In *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*, 2013.

[7] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for build debt: Experiences managing technical debt at google. In *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012.

[8] D. Sculley, M. E. Otey, M. Pohl, B. Spitznagel, J. Hainsworth, and Y. Zhou. Detecting adversarial advertisements in the wild. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, 2011.

[9] Securities and E. Commission. *SEC Charges Knight Capital With Violations of Market Access Rule*, 2013.

[10] A. Spector, P. Norvig, and S. Petrov. Google's hybrid approach to research. *Communications of the ACM*, 55 Issue 7, 2012.