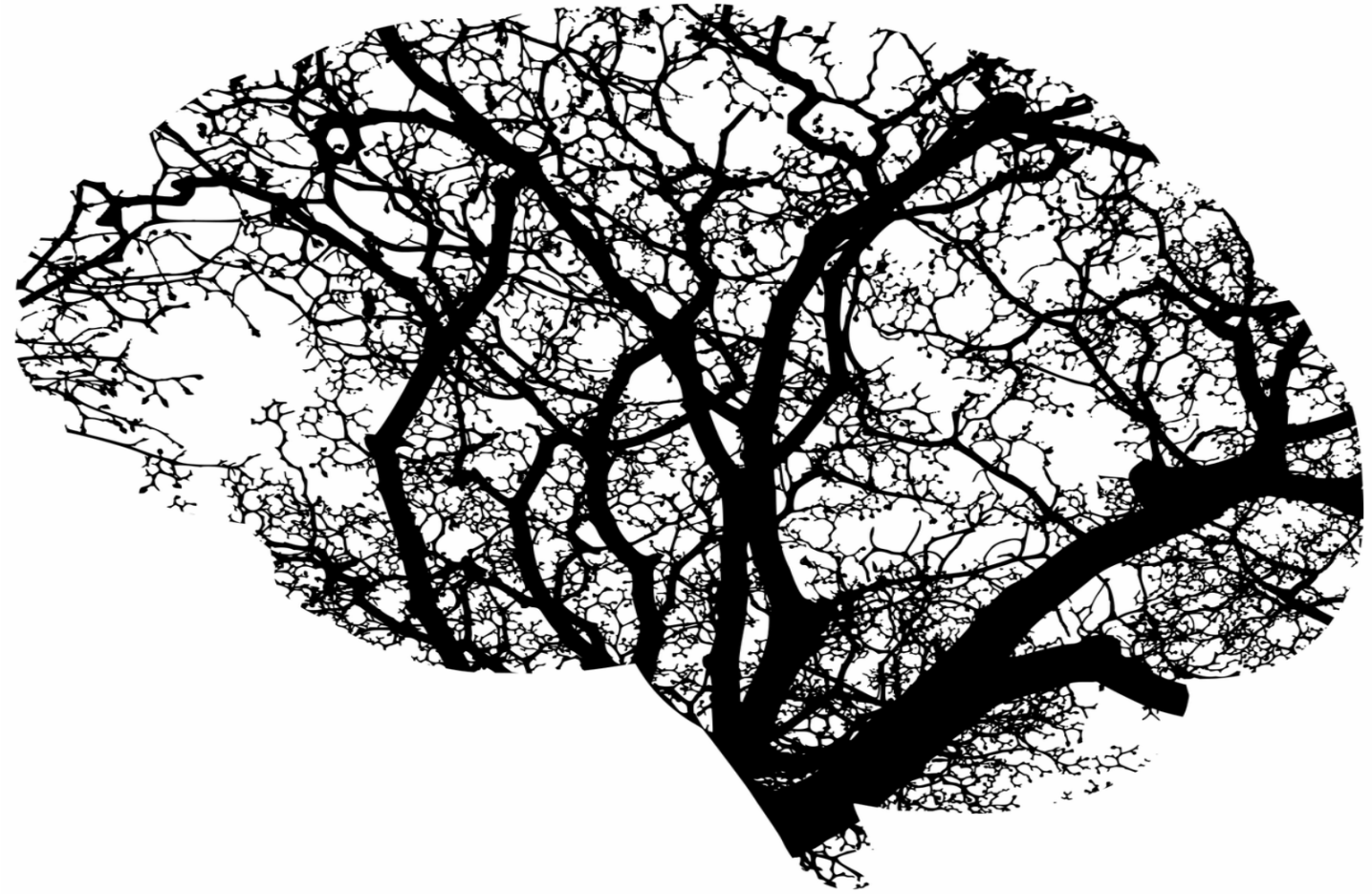


From shallow to deep learning in fraud

A Research Scientist's journey through hand-coded regressors, pickled trees, and attentive neural networks

Hao Yi Ong

Jul 10, 2018 · 9 min read

One week into my Research Science role at Lyft, I merged my first pull request into the Fraud team's code repository and deployed our fraud decision service. No, it wasn't to launch a groundbreaking user behavior activity-based convolutional recurrent neural network trained in a semi-supervised, adversarial fashion that challenges a user to prove her identity — it would be a couple of years before that. Embarrassingly, it was to remove a duplicate line of feature coefficients in a hand-coded logistic regression model rolled out a little less than a year before.

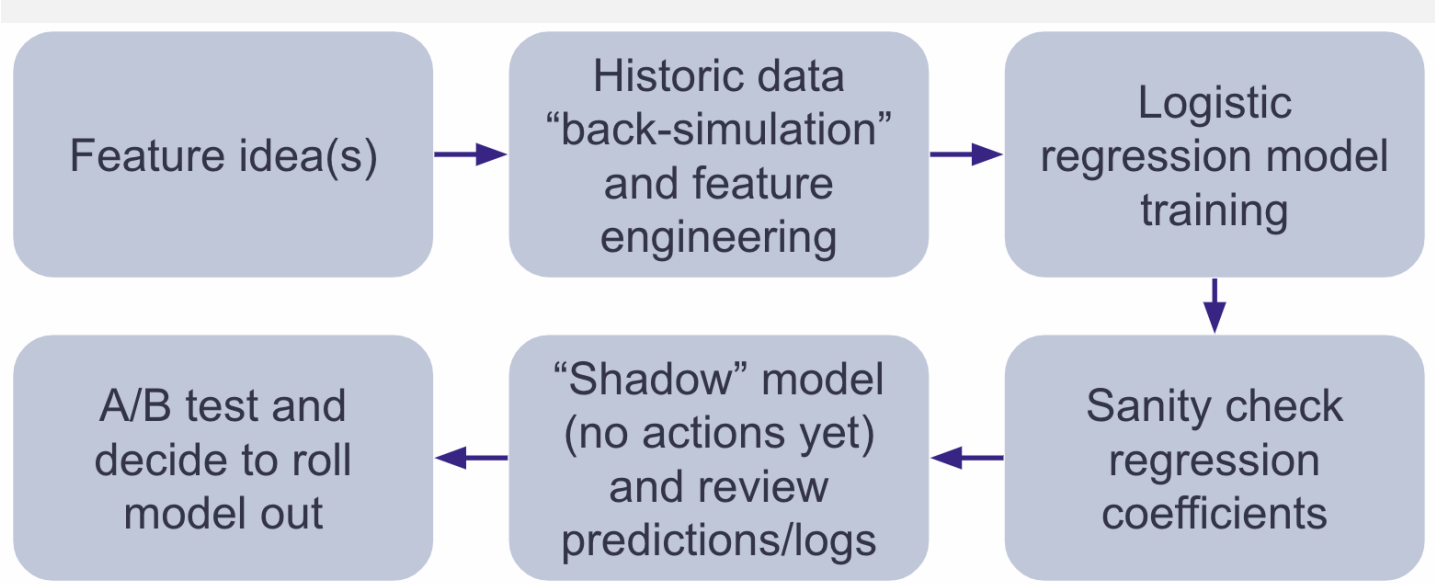
This small bug exposed a number of limitations of a system built primarily for a different type of usage — that of business rules that encapsulate simple, human-readable handcrafted logic. In our old worldview, models were simply extensions of business rules. As our models grew larger and features more complicated, model and feature definition inconsistencies between offline prototypes in Jupyter Notebooks and the production service occurred more frequently. These inconsistencies culminated in months-long back-and-forth cycles between Engineering and Research Science over feature definitions and ultimately blocked production. More worryingly, as we started exploring more modern, expressive machine learning libraries and frameworks like XGBoost and Tensorflow, it became clear that we needed drastic improvements to both the engineering stack and the prototype-to-production pipeline.

The redeeming point of this ordeal? That our model was robust to inflated feature values, which is important when faced with fast-adapting adversaries. But the overriding thought at that time was, *how can we improve our machine learning infrastructure?*

Bread and butter

In Fraud, we're primarily interested in classification algorithms that distinguish between good and fraudulent users on the platform. Despite the advent of more modern methods, logistic regression is still very much the bread-and-butter classifier of many industries. From a prototyping perspective, they are easy to implement and train, backed by theory, and amenable to mathematical analysis. From a practical standpoint, they are highly interpretable to even the layperson. For instance, the regression coefficients can be interpreted as how much a feature correlates with the likelihood of fraud. That makes diagnosing prediction errors easy: the key contributing features would have the largest summands in the logit. In terms of performance, a small team that constantly develops better features and maintains a good retraining pipeline will beat any huge team that manually handcrafts and manages hundreds of business rules.

At Lyft, a feature engineering-focused framework around such “classical” machine learning methods has worked well for a long time. It allowed us to keep a relatively simple codebase that was easy to iterate on, and we logged each prediction’s feature importances to debug and evaluate performance. More importantly, our simple models gave anyone on the team the ability to quickly prototype new features without a steep learning curve to the underlying technology. However, the simplicity of our linear models came at the cost of performance — we weren’t able to capture all of information that our features provided through linear correlations.

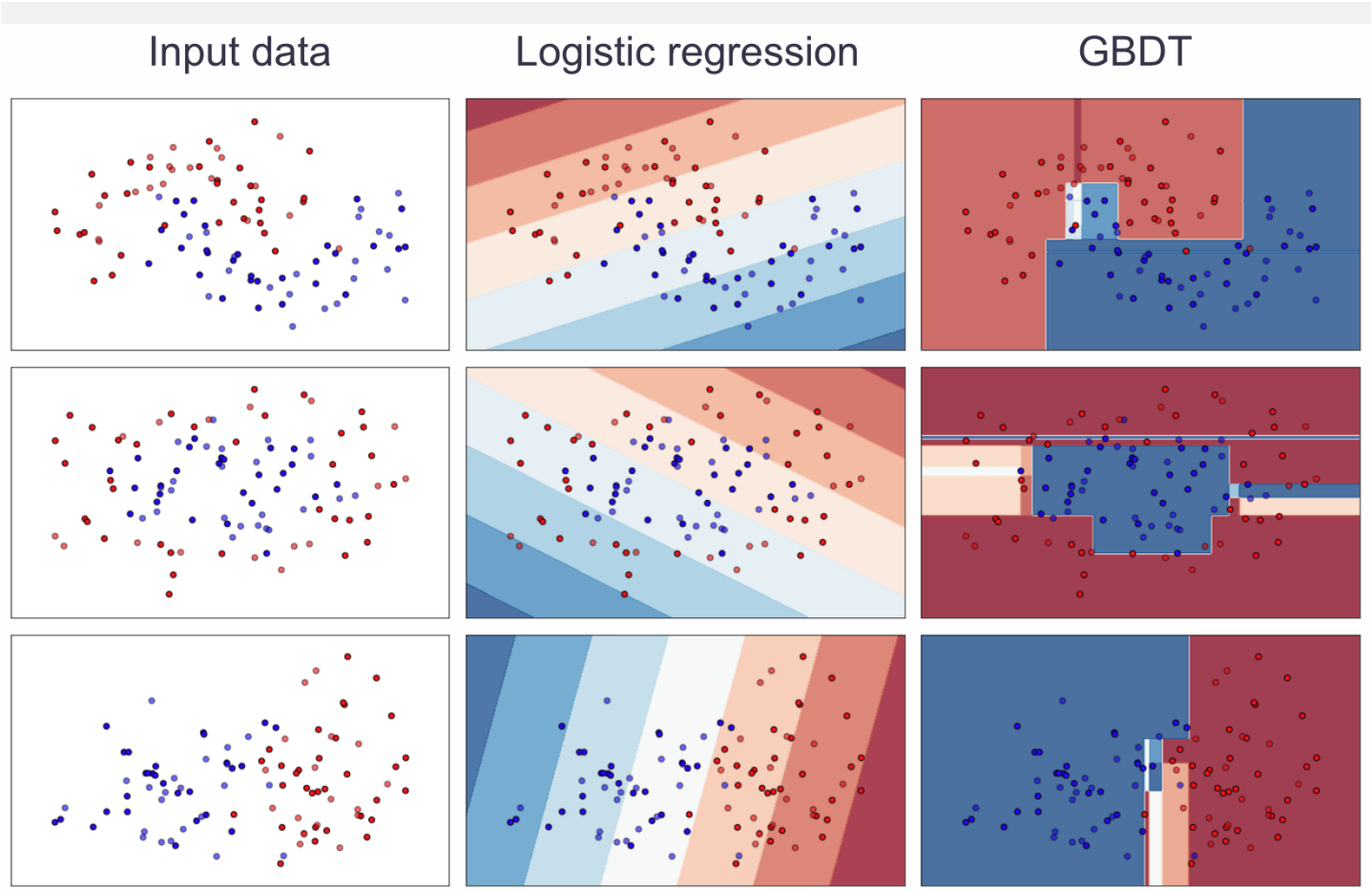


Our simple ML pipeline allowed anyone on the team to contribute through feature ideas.

Boosting performance

Any company that eventually grows large enough will invite second glances from fraudsters. Lyft is no exception. As the type of fraudsters that we attract grew in sophistication, we found it progressively harder to engineer features for the logistic regression model. Intuitively, the logistic regression model evaluates the information encoded in its features as a kind of weighted logical disjunction. To effectively capture any kind of higher-order interactions between features in the model, we’d have to hand-engineer them and add them as new features. For instance, it’s suspicious for a Lyft user with little ride history to suddenly start spending a lot on expensive rides. On the other hand, fraudsters “incubate” their accounts for a long while with cheaper rides before taking expensive rides in a short span of time. Encoding that type of complicated logic in a logistic regression model entails nontrivial feature acrobatics. We needed models that can capture such feature interactions more naturally.

After exploring several types of logistic regression and decision tree ensemble models, we settled on the gradient-boosted decision trees (GBDT) model trained on the popular XGBoost library given its ease of use and efficiency. At the theoretical level, we knew that the GBDT is more powerful than logistic regression. One instinctive way to understand the difference between the two is to visualize their decision boundaries. The logistic regression decision boundary is simply a hyperplane on the feature space, which means good features must exhibit strong linear relationships with the likelihood of fraud. For GBDT, the decision boundaries are smoothed versions of a collection of higher-dimensional boxes, which allow us to encode more complicated feature interactions. Indeed, we found that our GBDT had a relative precision gain by over 60% compared to the previous model for the same operating recall and feature set.



Gradient-boosted decision trees allow us to capture nonlinear relationships between feature values and labels.

Road to production

Based on impact alone, it would have been something like gross negligence not to replace our models with GBDTs. But there were a couple of complications.

For starters, ensemble models are generally harder to interpret and analyze compared to the simple logistic regression, and we needed to find viable methods to do both for practical reasons. While we sacrificed immediate model interpretability, we were able to find good replacements in boosted trees-specific feature importances and model explainers. The former are akin to the magnitude of regression coefficients and help us prioritize which features to productionize. The latter help us determine the features most influential for specific predictions and answer the (oft-feared) question of, *why was model X wrong?*

More importantly, we didn't have a good way to productionize our GBDTs. As seen in our opening example, even simple hand-coded logistic regression models are prone to bugs. Approaching GBDTs with thousands of decision trees and complicated branching logic would be a nightmare. What we needed was a simple and reliable way to serialize a prototype model on a Jupyter Notebook and load it onto a production system. We evaluated tools like PMML but found that most lacked support for bleeding-edge models and couldn't capture finer things like feature encoding and transforms in model pipelines. In the end, we built a library that utilizes the standardized scikit-learn API and pickle-based serialization. We knew of the many dangers of pickling and were thus extremely wary of the issues that may arise. But being even more wary of having to implement our own serialization scheme from scratch, we put additional guardrails around package requirements in a simple attempt at model versioning.

Presently, we use Tensorflow deep learning models in production because of their performance and their ability to work with signals that are hard to engineer features from. For instance, GBDTs do not gracefully handle sequential inputs like the histories of transactions, rides, and user activity on our app. (We'll discuss more of our latest models in an upcoming post.) Along the way, we've had to face other production issues and make adjustments to our internal serialization libraries and infrastructure. We also moved model execution and feature serving out of the original fraud decision service and built services around them that serves other teams' needs.

These changes pave a path for how we had long thought about elegantly handling model and package versioning. In our library, we didn't quite solve the package dependency versioning problem, which results from a mismatch between our development and production environments. That meant that we couldn't easily upgrade our "monolithic" decision service if we wanted to use newer package versions because our production models were based on older ones. I recall posing a naive question to the Fraud engineering

manager back in early 2017, *is it possible to spin up an EC2 instance with pip-frozen requirements for each model?*

Today, partly in response to our modeling needs, we're developing a more modern, container-based model execution that we believe puts a seamless prototype-to-production ML process within reach. But suggestions and questions like these didn't come naturally from research scientists back then.

Climbing over walls

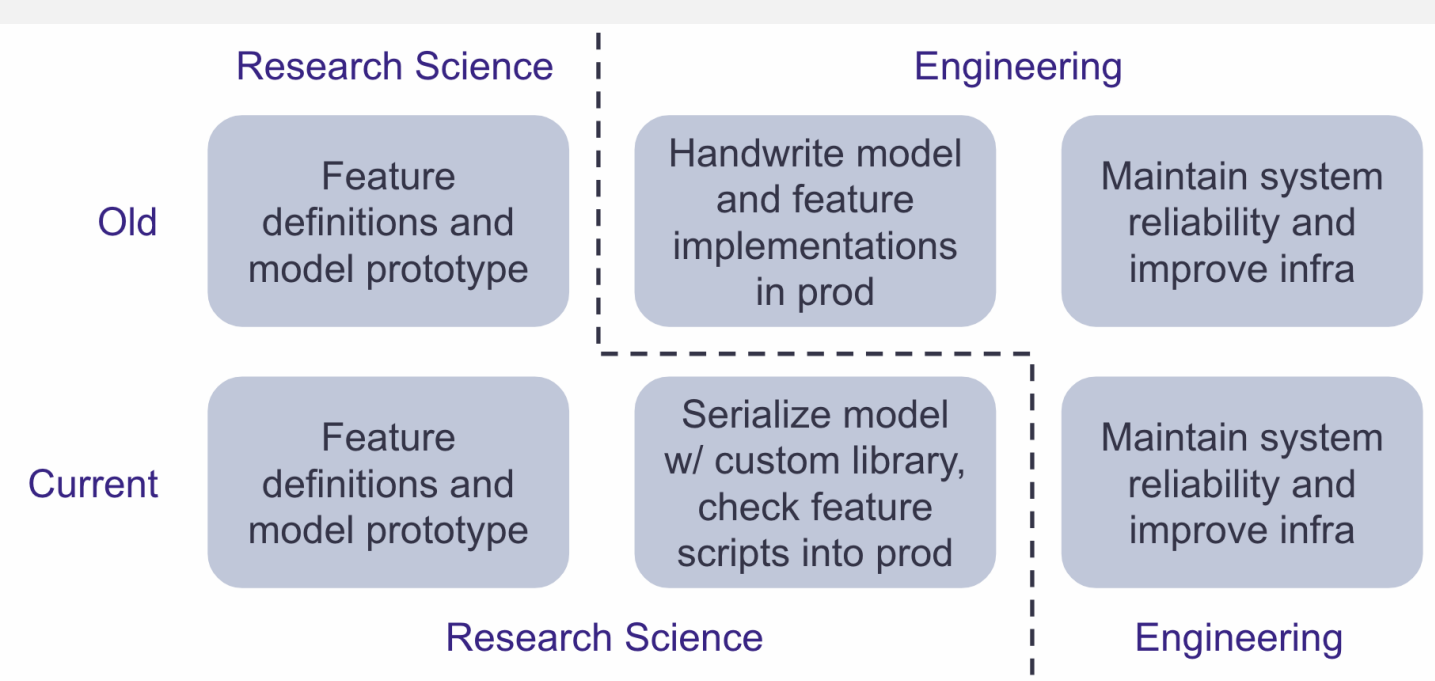
My foray into what may be more appropriately termed ML engineering in other companies started when it gradually dawned on me that there was a sharp lack of understanding on both Engineering and Research Science about each others' needs.

When I first joined Lyft, we had a siloed, "throw it over the wall" mentality where the scientist essentially stops work after the prototype feature set and model are built (hence the opening bug). For a while, it worked because the features were relatively simple. As our adversaries evolved and our features necessarily became more sophisticated, cracks begun to appear in that process. By the time I stepped foot into the company, "Feature War Rooms" (often in all-caps) to debug implementation inconsistencies had become a regular occurrence. It was then that I abandoned the old working model of handing off all of the feature and model production work to engineering.

Learning how the system worked wasn't exactly trivial and I quickly became *that annoying friend* to the Fraud Eng leads. But learning our microservice's capability and limits was valuable. Beyond writing my own features in the existing infrastructure, my working knowledge allowed me to contribute in designing better infrastructure. For instance, I redesigned our holdback and experimentation framework to measure the impact of our system in a hierarchical fashion. Previously, all our measurements were rule-based and it was impractical to evaluate our system on the whole. With today's framework, we can easily evaluate our system at the counter-fraud challenge and individual rule/model levels. I also contributed by building our first asynchronous model execution "trigger group." Unlike most of our models that run synchronously with strict execution time SLAs due to the product flow, I recognized that there were places where asynchronous execution unlocked the use of more complicated features and models.

Having a research scientist with good system knowledge helped guide and accelerate the engineering development process. Additionally, the change in work scope reduced

miscommunication between different roles and freed up engineers from rote feature implementation to focus more on, appropriately, building better platforms.



Model development past and now.

Dreaming ahead

We've come a long way from copy-pasting regression coefficients. Over the past two years, we've broken away from a monolithic fraud decision service that combined feature ingestion and preprocessing, model deserialization and execution, and counter-fraud challenges. Today, we're using and contributing to systems that provide the same team-agnostic services to a host of other teams that need the same sort of ML infrastructure.

But consolidating our ML infrastructure is just the first step. From automating feature "back-simulation" such that anyone has access to a trusted library of feature values to train on historical cases, to capitalizing on Dryft's powerful feature expressivity, to containerizing machine learning models as hinted above, we're building foundational blocks that ease the prototype-to-production workflow. Armed with these tools, we're developing automated feature generators that feed into model training, modular neural network layers that can be reused for different fraud problems, and even potentially bad ideas like protocols that allow us to collaboratively train and execute models across teams without direct access to the input features.

Further reading

If you enjoyed this post, follow and recommend! Also, don't be fooled into thinking that Research Science is only about ML or that Fraud Engineering is only about backend infrastructure; check out our other Research Science and Lyft Fraud posts!

- Interactions in fraud experiments: A case study in multivariable testing
- Stopping fraudsters by changing products
- What's in a name? The semantics of Science at Lyft

As always, Lyft is hiring! If you're passionate about developing state of the art machine learning models or designing the infrastructure that powers them, read more about our Research Science and Engineering roles and reach out to me!

This post was not just mine. Many thanks to Mike Ross, Steven Liu, Nick Chamandy, Josh Cherry, Will Megson, Yanshan Lu, Gil Arditi, Yaniv Goldenberg, Chris Elion, Ryan Lane, and Elaine Chow for reviews and edits!