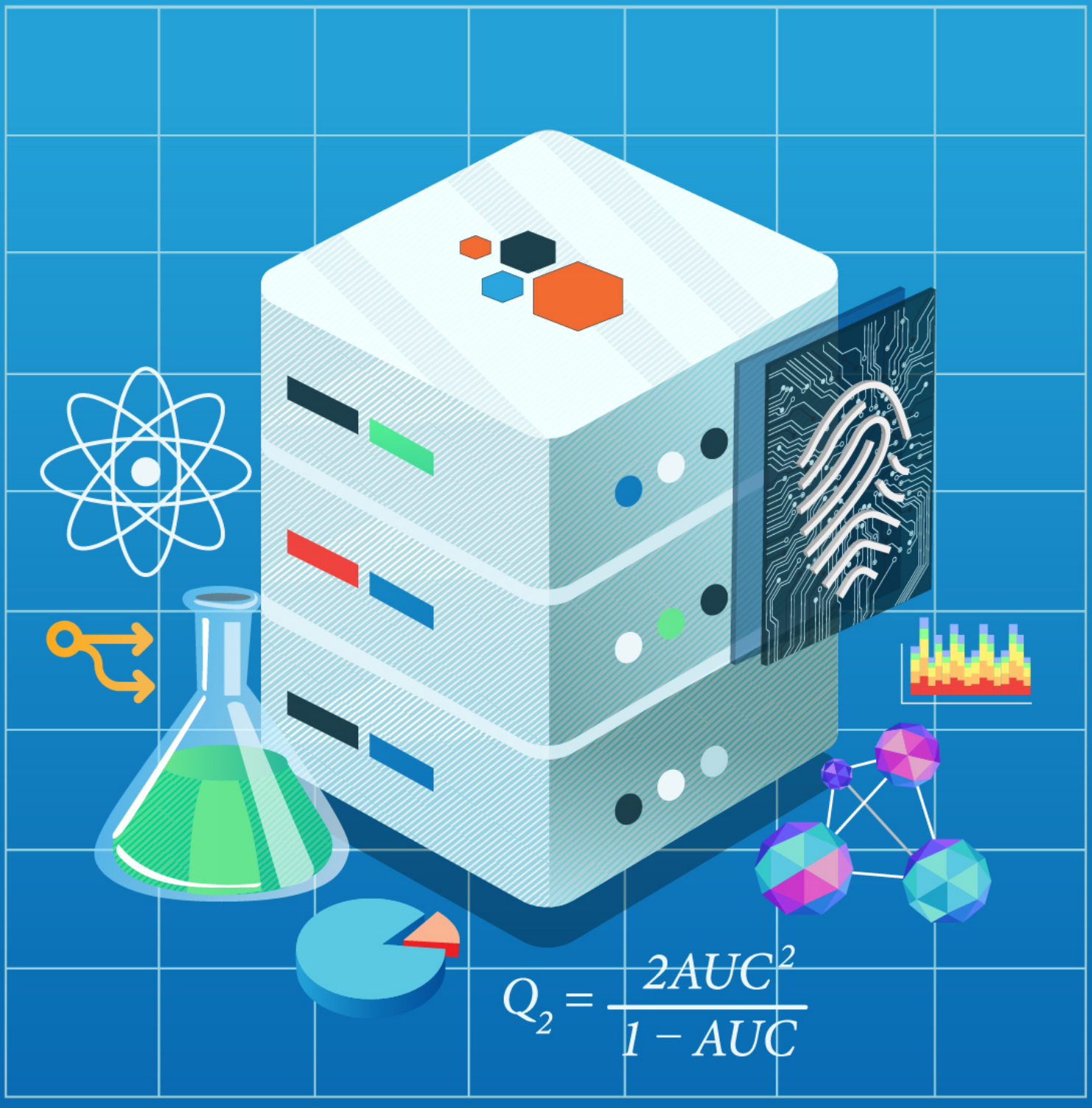
sift science

Running Effective Offline Machine Learning Experiments

By Alex Paino

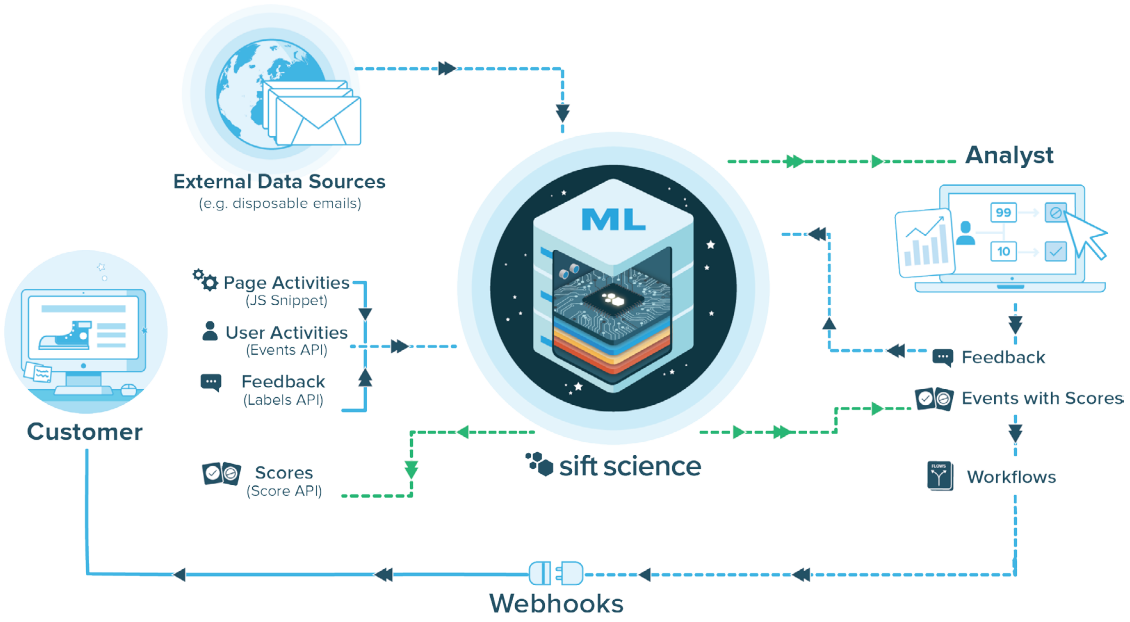


Introduction

At Sift Science we use machine learning to prevent various forms of abuse on the internet. To do this, we start with some data provided by our customers:

1. Page view data sent via our Javascript snippet
2. Event data for important events, such as the creation of an order or account, sent through our Events API
3. Feedback through our Labels API or our web Console

We then combine this data with a few external data sources and use it to train a variety of models to produce scores for 4 distinct abuse prevention products (payment, promotion, content, and account), which are then accessible through Workflows and our Score API:

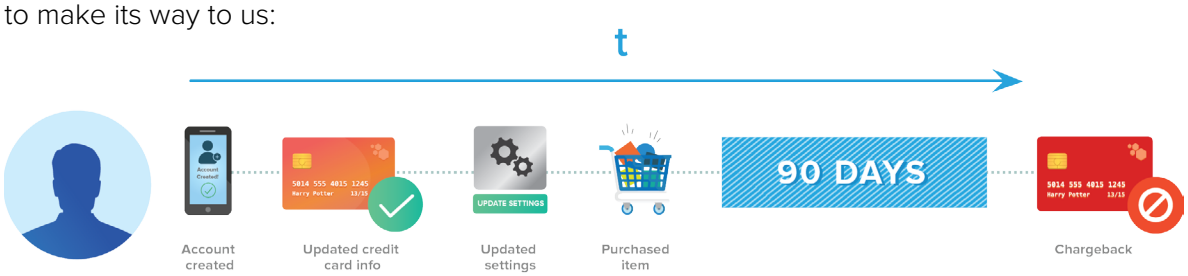


Improving the accuracy of these scores leads directly to more revenue, less fraud, and better end-user experiences for our customers. Therefore, we spend a large amount of effort trying to improve our machine learning's accuracy. In this article on ML Experiments at Sift, we'll discuss some lessons we have learned around building an ML experimentation framework that allows us to do this effectively. We'll start by describing how we minimize bias in offline experiments.

Part 1: Minimizing Bias

Running experiments correctly

A useful experimentation framework must allow its user to quickly run an experiment and gather its results while introducing as little bias as possible. At Sift, this is particularly difficult because of the long feedback delay we typically see. The worst case of this is with payment fraud, where feedback is usually in the form of a chargeback, which can take **up to 90 days** to make its way to us:



Outside of payment abuse, our other abuse products still typically require hours or days to receive feedback. Because of this, we **can't use live A/B tests** to measure the effect of a change to our system.

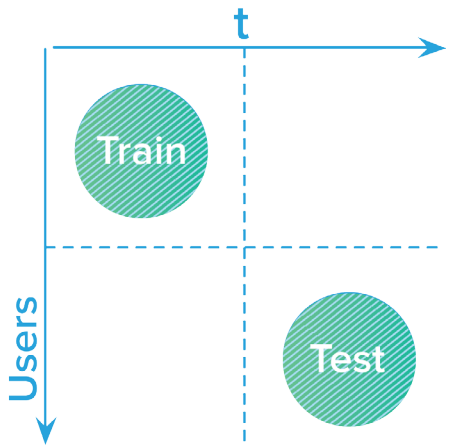
The problem then becomes: how can we run offline experiments that correctly simulate the live case? Below we present some of the issues that make this a difficult problem along with the approach we have taken to address them.

Train-test set creation

The first challenge we faced when constructing our experiments was determining how we should construct our training, validation, and testing sets (each of which consists of a collection of user Events which we would have scored in the online case). While doing this, we found:

- **Sets need to be disjoint in time.** One of the strongest aspects of our ML is its ability to accurately connect new accounts to accounts previously known to be associated with fraud. Therefore, we require that all data in the training set occurred prior to the validation set, and that the validation set occurred prior to the testing set.

- **Sets also need to be disjoint in user-space.** We don't want to give ourselves credit for correctly identifying known fraudulent accounts as being fraudulent, as this provides no value to our customers, so we need to ensure that each end-user only appears in a single set. This gives us a train-test set split that looks like this:

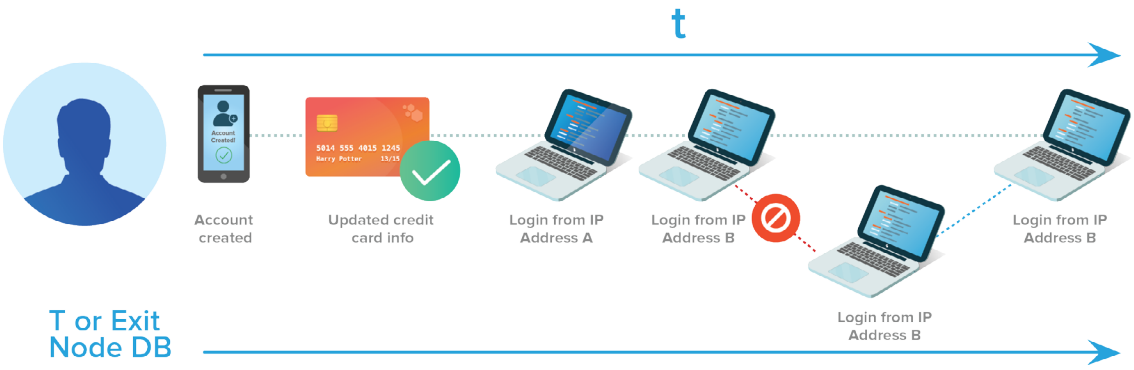


- **Class skew is problematic.** Our typical customer only sees fraud or abuse in roughly 1/50th of all transactions/posts/etc., so to ensure we don't train something close to the always-negative classifier we have to heavily downsample the not-fraud class.

Preventing cheating

A challenge that we continually face is preventing our models from "cheating" in offline experiments by using information that would not be available to our live system (e.g. anything from the future). A simple case of this was already discussed above when we talked about our train/validation/test sets being disjoint in time. However, this problem surfaces itself in more subtle ways in our system, which has led us to take the following precautions:

- **External data sources need to be versioned.** We leverage a number of third-party data sources in our system, such as IP-Geo mappings and open proxy lists. When running offline experiments, we need to ensure we only ever use the latest version of these data sources available at the point in time at which we are extracting features for a user:



- **Verify all new data sources are not biased in any way by ground truth.** In addition to ensuring external and derived data sources do not leak backwards in time, we also need to make sure the collection of these data sources is not a function of the labels we have collected. For example, when backfilling some new data source from historical data, we can't bias the backfill process towards more aggressively backfilling data for users belonging to the positive (i.e. fraudulent) class. We have run into this exact issue in the past when experimenting with third-party social data sources, which led to some too-good-to-be-true initial results. Since then, we have learned to more thoroughly vet all new data sources introduced into our pipeline, including the more recent introduction of an email age feature which is derived from our customers' data.

Parity with the live system

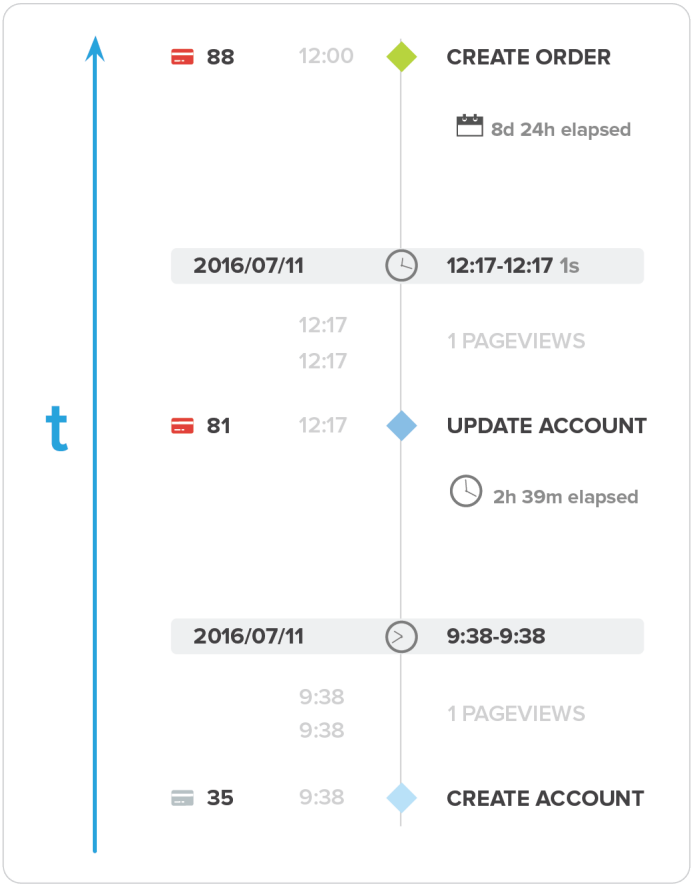
Closely related to preventing cheating in an offline system, we've also put special effort towards maintaining parity between the offline and live systems whenever possible. The simplest lesson here is to aim to reuse the same code paths online and offline; this helps to minimize the surface area where potential bugs and biases could creep into the offline system.

However, creating an offline simulation necessitates a large amount of code that won't be shared with the online system, such as the machinery required to play back all feedback in the proper order to support online learning. We've found that parity tests, as well as an easily decomposable pipeline, can be quite useful in ensuring the offline system behaves as intended.

Measuring value to customers

Beyond ensuring the models are trained and evaluated in a correct manner, we also need to pay attention to how closely our evaluation techniques match how our customers use our product. For our Payment Fraud product, this means we only evaluate our ML using the scores given at time of checkout, while for the Content Abuse product we only use the scores given at the time of a post being created.

For example, consider the following sequence of events and scores:



If in this case the customer used the score at time of the Create Account event to determine if the user should be allowed to create the account, then the score of 35 is the one we should use to evaluate our accuracy. That is, if the user eventually ended up being labeled with a positive Account Abuse label, then we should not get credit for giving the user a higher score after the Create Account event. This ensures a positive change in our evaluation metrics actually correlates to a positive change for our customers.

Part 2: Analyzing Thousands of Models

Previously, we described how we minimize bias in our offline experimentation framework to ensure we run representative experiments. Once this is achieved, the next step is to determine how to compare the results of two separate experiments. At Sift this is non-trivial because:

1. **Every Sift Score is a function of several distinct models**. The final score is produced by an ensemble model which combines the output from multiple globally-scoped models and multiple customer-scoped models. Here, "globally-scoped" means "trained over all of our customers' data".
2. We produce **4 different types of Sift Scores** -- one for each abuse prevention product.
3. We have **thousands of customers** which we would like to include in our experiments. Each of these customers has their own customer-specific models, complete with their own custom features, class skew, and noise levels.

Put together, we have **tens of thousands of different models** we could potentially evaluate. Clearly, it is not feasible to examine 10,000+ ROC plots, F1 scores, etc.; we need some way of summarizing these evaluations.

Below we describe a few specific lessons we've learned in this area along with their original motivation.

We need to evaluate our models in a threshold-agnostic manner

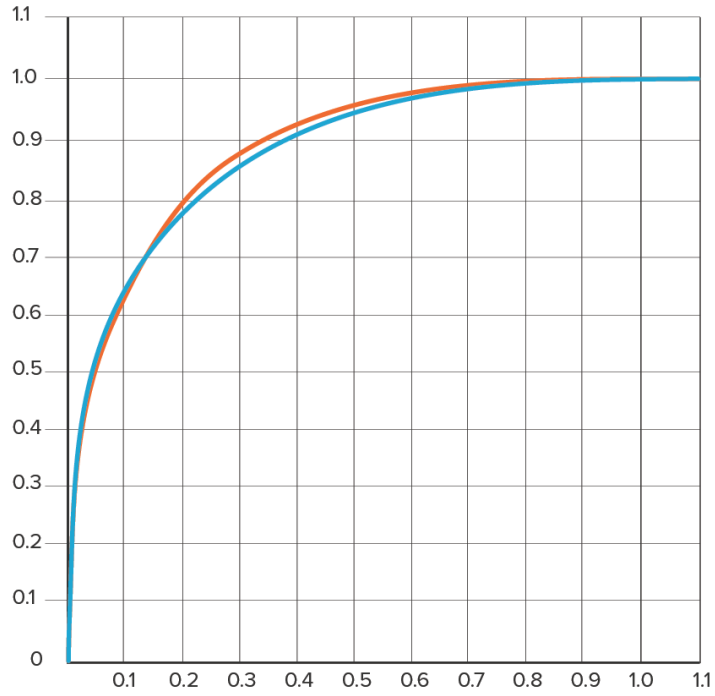
Problem: threshold-based evaluation metrics misrepresent our models' accuracy

All of the models we train are used as soft classifiers to produce scores in the range [0, 1], where 1 indicates a very high likelihood of fraud, and 0 indicates a very low likelihood of fraud. Instead of thresholding these soft predictions on our end, we surface them directly to our customers, who may use them in a number of ways. A common use case is to pick two thresholds: one that determines when a user is blocked, and one that determines when a user is reviewed. However, more advanced customers may leverage additional thresholds to alter other aspects of their product; for example, some customers will present a more streamlined checkout process to users with very low scores. Because of this diversity in use cases, we cannot use any evaluation metrics that are a function of a fixed threshold when evaluating our models. This includes raw precision, recall, and false positive rate metrics, as well as any metrics derived from them (such as F-1 score).

Solution: use evaluation metrics that are not tied to a specific threshold

Our solution is to assess our models in a threshold-agnostic manner by measuring the probability of a bad user being assigned a higher score than a good user. To do this, we construct a receiver operating characteristic (ROC) curve and measure the area under it. While this area is very useful for summarizing the accuracy of a model with a single metric, we have also found that examining the ROC curve itself can provide further insight into how a new model is better than an existing one, again in a threshold-agnostic manner.

For example, in the following plot we see the ROC curve for two competing models:



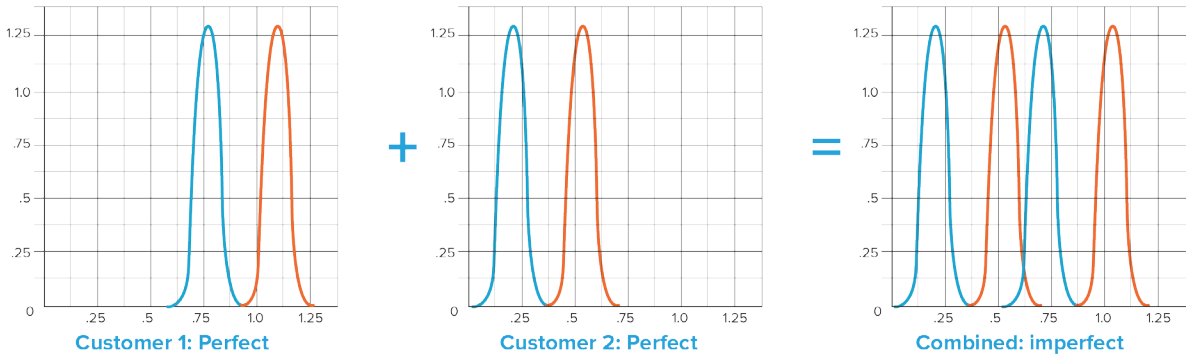
The model represented by the red ROC curve has a greater ROC AUC than the model represented by the blue ROC curve. However, by examining the actual curves we can see that all of the improvement occurs at operating points with false positive rates greater than 0.1, and that below that point the model in blue is actually better. If the customer in question happened to only operate with a false positive rate less than 0.1, as is common for our customers, then the experiment that yielded the red ROC curve would not result in an improvement for them.

We need to do partition score samples by customer

Problem: combining samples across customers leads to misleading results

A natural way to evaluate one of our models would be to take all scores it produced during evaluation and measure the resulting ROC AUC. However, we found that this approach leads to misleading evaluation results in our case as it makes the assumption that the relative ordering between any pair of scores is equally important; this is not true for us since these scores are coming from a number of different customers, and we only care about the relative ordering of scores within a customer. That is, if a score given to a fraudster from customer A is less than the score given to a good user from customer B, this should not count against our system, as it does not say anything about the accuracy customers A or B will see.

If all of our customers had identical operating parameters, integrations, and fraud problems, this would not be an issue. Of course, this is not the case -- our customers have a wide range of integrations unique to their vertical, use case, and other restrictions. For example, some of our customers are unable to integrate with our Javascript snippet, which generally results in a "dampening" effect on their score distribution -- i.e., the separation between the positive and negative distributions is reduced due to a lack of high-precision signal. Combining score samples from such a customer with those from a customer that is integrated with our Javascript snippet can lead to our evaluation portraying a change that is positive for both individually as being negative in the aggregate. An extreme version of this case is outlined in these graphs:



Customer 1: Perfect | Customer 2: Perfect | Combined: imperfect

Solution: focus on summary metrics from each customer

Ever since we had some experiments fall victim to this problem, we have been sure to partition score samples by customer prior to applying any binary classification metrics. This, of course, leads to many more summary metrics that need to be considered, which motivates our final lesson of the day...

Statistical tests help to surface the most important results of an experiment

Problem: there is no good way to collapse summary metrics across customers

Given a collection of ROC AUC metrics for all customers, we need some way of boiling them down to a small handful that could all reasonably be examined by the engineer running the experiment. A natural first attempt would be to take a (possibly weighted) average over these metrics. We tried this initially with both volume-weighted and unweighted averages. However, we quickly found that neither worked that well:

1. Unweighted averages were too noisy, as we have many customers with just a handful of positive samples
2. Volume weighted averages tended to be dominated by the very largest customers, such as Airbnb, and would hide significant improvements for our medium-small customers.

Solution: use statistical tests to filter through results

The most helpful lesson we've learned when it comes to consolidating experiment results is to lean on statistical tests wherever possible. By aggressively applying tests to more granular metrics, we are able to ensure that only interesting comparisons are surfaced when evaluating an experiment.

We apply this lesson to the collection of per-customer ROC AUC metrics mentioned previously by producing confidence intervals around each metric using the following formula:

$$SE(AUC) = \sqrt{\frac{AUC(1-AUC) + (N_1-1)(Q_1-AUC^2) + (N_2-1)(Q_2-AUC^2)}{N_1 N_2}}$$

$$Q_1 = \frac{AUC}{2 - AUC}$$

$$Q_2 = \frac{2AUC^2}{1 - AUC}$$

This formula is described in more detail here. Constructing confidence intervals in this manner allows us to quickly filter out all individual comparisons that are not significantly changed at the 95%, 90%, etc., confidence level. By incorporating these confidence intervals into our Experiment Evaluation page (to be described in the next post in the series), we are able to quickly find which (customer, abuse type, model) combinations are most significantly affected by an experiment. From there, we can then view individual ROC curves to get a more in-depth sense of the effect of an experiment.

Part 3: Building the right tools

We've already discussed how we ensure correctness when conducting and analyzing machine learning experiments at Sift Science. Determining how to construct experiments and evaluate their results correctly is great, but it is only useful if you ensure all of the experiments you run and evaluate use these correct techniques. If running an experiment correctly requires re-inventing all of the logic put in place to account for issues such as those discussed in the previous posts, then odds are that your team will not always run correct experiments.

Our solution to this problem is to provide **tools that bake in correctness** and make it easy for all experiments to be run and analyzed correctly. We have found that doing this has allowed our entire engineering organization to be more productive, as:

a) Less time is wasted running and analyzing experiments with invalid results

b) Engineers not on our machine learning team can easily and safely run experiments (e.g. an infrastructure engineer can easily experiment with improvements that speed up training or allow us to use more data)

The tools we have built allow us to:

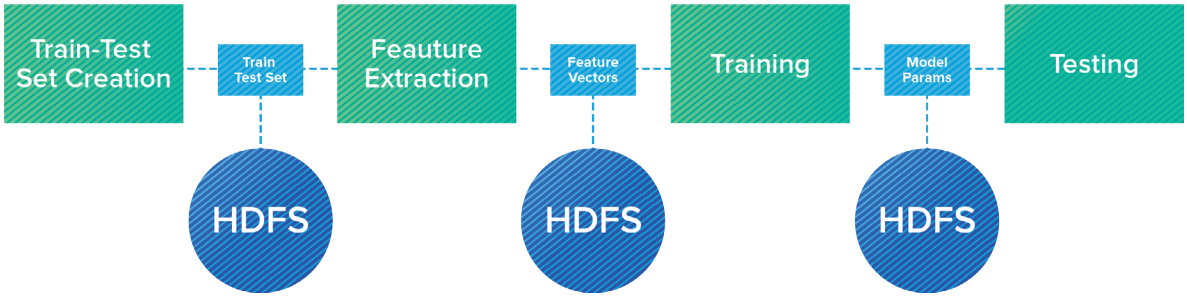
1. Easily run correct experiments
2. Quickly perform high-level analysis of an experiment's results
3. Dig deeper into specific aspects of an experiment

Each of these is described in more detail below.

Tools that make it easy to run correct experiments

We have spent a great deal of time creating an offline training pipeline that bakes in all of the lessons we have learned around conducting experiments correctly. Today, using this pipeline is the only accepted way of producing valid experiment results at Sift.

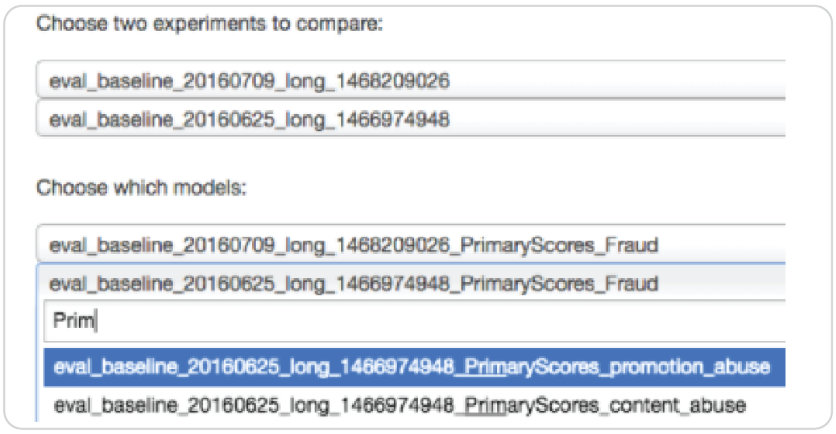
Our pipeline consists of 4 primary phases:



Each of these phases writes out intermediate results to HDFS, which allows us to easily reuse them to save time in subsequent experiments. This allows, e.g., the reuse of previously extracted feature vectors in a new experiment that only modifies the training stage. To make running experiments as easy as possible, we additionally have scripts in place that automatically setup the proper dependencies and orchestrate all jobs in the pipeline.

High level analysis tools

The primary tool we use to quickly analyze the results of an experiment is our Experiment Eval tool. We begin an analysis using this tool by selecting two experiments and an abuse type (e.g. payment abuse):



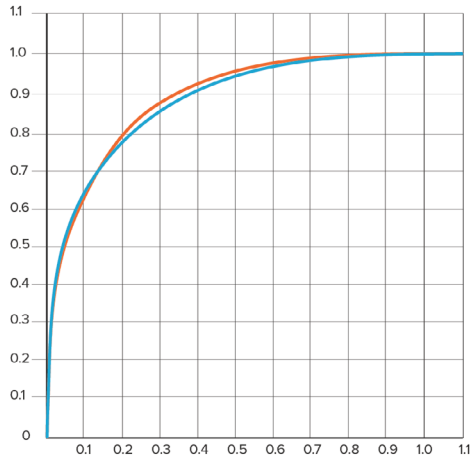
From here we can quickly see all customers affected by the selected experiment sorted according to the significance of the change in their ROC AUC:

[redacted] has better auc for **8** significant customers

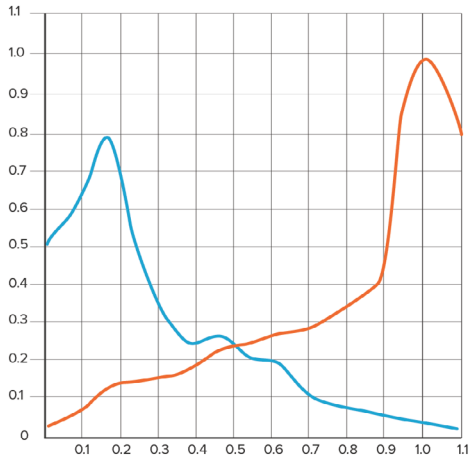
[redacted] has better auc for **6** significant customers

Customer	AUCs for [redacted]	AUCs for [redacted]
[redacted]	0.987	**0.992**
[redacted]	0.316	**0.875**
[redacted]	**0.917**	0.828
[redacted]	0.941	**0.957**
[redacted]	0.892	**0.915**
[redacted]	**0.985**	0.911
[redacted]	**0.390**	0.171
[redacted]	0.791	**0.927**
[redacted]	**0.934**	0.863
[redacted]	**0.929**	0.893
[redacted]	**0.949**	0.873
[redacted]	**0.927**	0.790

This allows us to quickly filter out the thousands of customers who are not significantly affected by the experiment under analysis. Given this, we can then dig into the (usually) small handful of customers who actually are affected. The Experiment Eval page also makes this easy to do by allowing us to surface the ROC, PR, and score distribution plots for a specific customer:



ROC



Score distribution

This mode also gives us various summary statistics, such as weighted mean error, the best F1 score, ROC AUC, etc. Together, these stats and plots are usually sufficient for understanding the nature of a change to a customer's accuracy.

Tools for fine-grained analysis

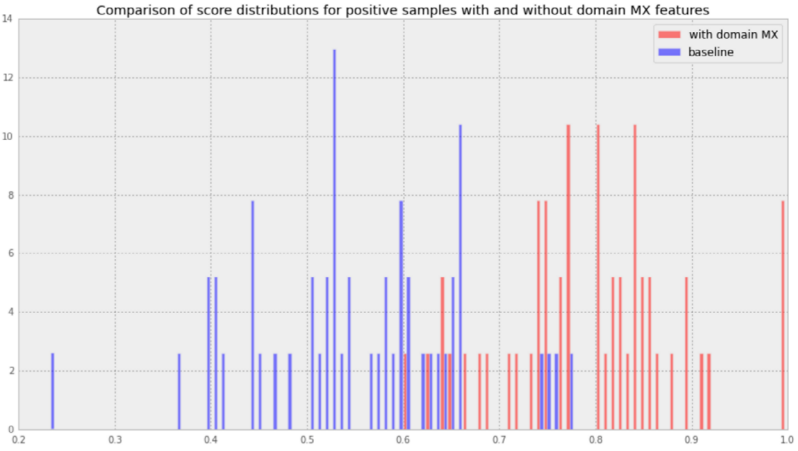
For more complicated analysis, we have found it necessary to support some tools that allow us to drill more deeply into an experiment. For this use case, **we've found Jupyter notebooks to be a perfect fit.** These notebooks operate on side-outputs from our offline experimentation pipeline, and allow us to dig into the particulars of a given experiment by, e.g., restricting our analysis to only those samples that include a newly introduced feature.

One example where these notebooks have been useful was when we experimented with features derived from the MX record associated with the domain of a user's email address. When we first analyzed this experiment using our Experiment Eval tool, we did not observe any significant difference in accuracy due to this change. But our intuition told us it should be adding value, so we dug deeper using Jupyter to find some users who would be affected by these new features, and sure enough, were able to find an improvement:

```
In [ ]: baseline_user_scores = []
        domain_mx_user_scores = []

        for uid in fraud_users_with_domain_mx:
            key = cid + ":" + uid
            baseline_user_scores.append(baseline_scores_rdd.lookup(key)['payment_abuse'])
            domain_mx_user_scores.append(domain_mx_scores_rdd.lookup(key)['payment_abuse'])
```

```
In [37]: plt.hist([domain_mx_user_scores, baseline_user_scores], 100, normed=1, alpha=0.5, color=["red", "blue"])
         plt.title("Comparison of score distributions for positive samples with and without domain MX features")
         plt.legend(["with domain MX", "baseline"])
         plt.show()
```




```
In [36]: print(np.mean(baseline_user_scores))
         print(np.mean(domain_mx_user_scores))
```

```
0.549589552871
0.795243473271
```

Our Experiment Eval tool described above did not pick up on this change because it only affected a small number of users; however, the small number of users it did affect were causing problems for one of our customers, and adding these features ended up helping them quite a bit.

About Alex Paino

Alex Paino received his Bachelors in Mathematics and Computer Engineering from the University of Missouri, where he focused on machine learning and computer vision research. At Sift Science, Alex works as a software engineer, focusing primarily on machine learning modeling.

Contact

scientists@siftscience.com
www.siftscience.com