An Architecture for Agile Machine Learning in Real-Time Applications

Johann Schleier-Smith
if(we) Inc.
848 Battery St.
San Francisco, CA 94111
johann@ifwe.co

ABSTRACT

Machine learning techniques have proved effective in recommender systems and other applications, yet teams working to deploy them lack many of the advantages that those in more established software disciplines today take for granted. The well-known Agile methodology advances projects in a chain of rapid development cycles, with subsequent steps often informed by production experiments. Support for such workflow in machine learning applications remains primitive.

The platform developed at if(we) embodies a specific machine learning approach and a rigorous data architecture constraint, so allowing teams to work in rapid iterative cycles. We require models to consume data from a time-ordered event history, and we focus on facilitating creative feature engineering. We make it practical for data scientists to use the same model code in development and in production deployment, and make it practical for them to collaborate on complex models.

We deliver real-time recommendations at scale, returning top results from among 10,000,000 candidates with sub-second response times and incorporating new updates in just a few seconds. Using the approach and architecture described here, our team can routinely go from ideas for new models to production-validated results within two weeks.

Categories and Subject Descriptors

H.4.m [**Information Systems Applications**]: Miscellaneous

Keywords

Agile; Recommender Systems; Machine Learning

1. INTRODUCTION

Innovative companies often use short product cycles to gain advantage in fast-moving competitive environments. Among social networks, Facebook is known for especially frequent release cycles [11], and if(we) counts such capabilities as crucial to the early success of the Tagged and hi5 web sites, which today form a social network with more than 300 million registered members. We especially value the quick feedback loop between product ideas and production experiments, with schedules measured in days or weeks rather than months, quarters, or years.

Today, on account of the approach described here, if(we) can develop and deploy new machine learning systems, even real-time recommendations, just as rapidly as we do web or mobile applications. This represents a sharp improvement over our experience with traditional machine learning approaches, and we have been quick to take advantage of these capabilities in releasing a stream of product improvements.

Our system puts emphasis on creative feature engineering, relying on data scientists to design transformations that create high-value signals from raw facts. We work with common and well understood machine learning techniques such as logistic regression and decision trees, interface with popular tools such as R, Matlab, and Python, but invest heavily in the framework for data transformation, production state management, model description, model training, back-testing and validation, production monitoring, and production experimentation,

In what we believe to be a key innovation not previously described, our models only consume data inputs from a time-ordered event history. By replaying this history we can always compute point-in-time feature state for training and back-testing purposes, even with new models. We also have a well-defined path to deploying new models to production: we start out by playing back history, rolling forward in time to the present, then transition seamlessly to real-time streaming. By construction, our model code works just the same with inputs that are months old as with those that are milliseconds old, making it practical to use a single model description in both development and production deployment.

In adopting the architecture and approach described here, we bring to machine learning the sort of rapid iterative cycles that are well established in Agile software development practice [28], and along with this the benefits.

Our approach can be summarized as follows:

- **Event history is primary:** Our models consume data inputs as time-ordered event history, updating model signals represented by *online features*, state variables for which we have efficient incremental update routines with practical storage requirements.

KDD'15, August 10-13, 2015, Sydney, NSW, Australia.
ACM 978-1-4503-3664-2/15/08.
DOI: http://dx.doi.org/10.1145/2783258.2788628.

- **Emphasis on creative feature engineering:** We rely heavily on the insights of data scientists and on their ability to devise data transformations that yield high-value features (or signals) for machine learning algorithms.

- **One model representation:** The same code used by data scientists during model development is deployed to production. Our models are written in Scala, which makes reasonably efficient implementations practical, which offers advanced composability and rich abstractions, and which allows our software engineers to create library code providing a DSL-like environment, one where data scientists can express feature transformations in a natural way.

- **Works with standard machine learning tools:** There exists a tremendous variety of sophisticated tools for training machine learning models. Our approach interfaces cleanly with R, Matlab, Vowpal Wabbit, and other popular software packages.

Key benefits include:

- **Quick iterations:** We routinely need just a few days to go from an idea, a suggestion for a feature that might have increased predictive power, to production implementation and experimental verification. This rapid cycle keeps the cost of trying new ideas low, facilitates iterative progress, and helps data scientists stay engaged and focused.

- **Natural real-time processing:** Although the event history approach can be used in off-line or batch scenarios, using real-time signals carries no additional cost. Even in applications with relaxed update requirements, eliminating batch processing can make a problem easier to reason about. Also, real-time stream processing usually allows for more uniform production workload, which can be easier to manage.

- **Improved collaboration:** In many corporate environments data scientists are inclined to work in silos, and they commonly find it difficult to reproduce one another's work [16]. Our environment offers data scientists the best enablers of modern software development, including version control, continuous integration, automated testing, frequent deployment, and production monitoring. With these tools, plus shared access to a production event history, it becomes much more natural for data scientists to collaborate as they solve problems.

The primary driver of our work has been building the recommendation engine for a dating product, Meet Me, that is offered within Tagged and hi5. Choosing from roughly 10 million active members, and incorporating signals from recent seconds, the system produces an ordered list of profiles to be presented to the user for consideration. We devote Section 2 to a detailed description of this application and the design choices that it spurred. The open source Antelope framework, described in Section 3, generalizes the concepts developed for Meet Me, adds usability improvements, and provides a reference implementation independent from the if(we) codebase and applications.

2. THE MEET ME DATING APPLICATION

2.1 Problem

Among various features offered by the Tagged and hi5 social platform, Meet Me caters most directly to those seeking romantic connections; it serves as the center of the dating offering. The user is presented with one profile at a time, and prompted with a simple question, "are you interested?" In response, the user may select from two options, variously labeled as *yes* or *no*, ✓or ✗. In our terminology, we describe such a response as a *vote*, and in this discussion we standardize on the response terminology *positive* and *negative*. Two users achieve a *match* when both express mutual interest through positive votes for one another, and creating such matches is an important optimization goal for our algorithm. An illustrative screen shot of the Meet Me user interface appears in Figure 1.

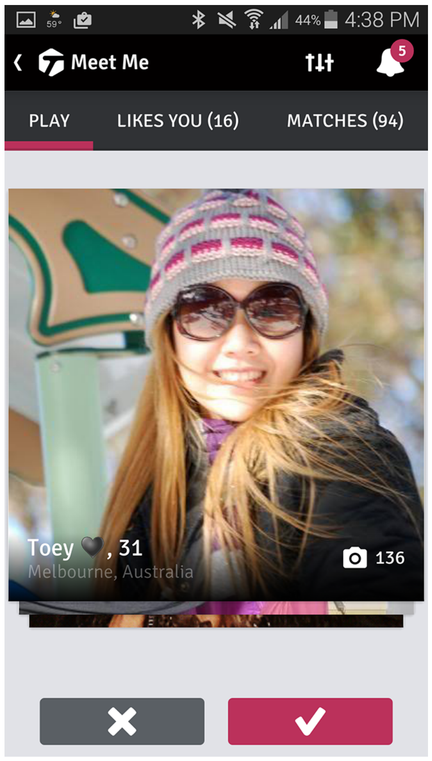


Figure 1: The Meet Me voting interface, shown here in the Tagged Android application. Users can touch the voting buttons at the bottom of the screen, or may swipe the presented profile towards the right to register a positive vote, or towards the left to register a negative vote.

The Meet Me style of matching, adopted by Tagged in 2008 and in 2012 by hi5, following a merger, appears to have been introduced by Hot or Not Inc. in the early 2000s. In recent years it attracted even greater attention as embodied in the Tinder mobile dating app. Notable implementations also include those by online social discovery companies Badoo, a UK company headquartered in London, and MeetMe Inc., a US company headquartered in New Hope, Pennsylvania.

We can view our Meet Me optimization problem from one of several perspectives, but prefer a formulation from the viewpoint of the user, posing the problem as follows: "given the millions of active user profiles matching basic eligibility criteria (principally filter settings), which should we next select to show?" We believe that focusing on the user helps data scientists build empathy for the individual experience, which is an important guide to intuition. It is our conjecture and our hope that separately optimizing for individual users produces a result that is well optimized for all users. Still, in developing a model for the experience of one user, we must account for behavior of other users as well. Most obviously, we recognize that it is not sufficient to derive recommendation from predictions of profiles that a user is likely to be interested in, it is also important that interest is reciprocated and mutual.

We decompose the problem by expressing the match probability in terms of separate conditional probabilities:

$$\begin{aligned} p(match_{a\leftrightarrow b}|vote_{a\to b}) &= p(vote^{+}_{a\to b} \wedge vote^{+}_{b\to a}|vote_{a\to b}) \\ &= p(vote^{+}_{a\to b}|vote_{a\to b}) \times p(vote_{b\to a}|vote^{+}_{a\to b}) \\ &\quad \times p(vote^{+}_{b\to a}|vote_{b\to a} \wedge vote^{+}_{a\to b}) \end{aligned} \tag{1}$$

where $match_{a\leftrightarrow b}$ represents a match between user a and user b, $vote_{a\to b}$ represents a vote, either positive or negative, by user a on user b, and $vote^{+}_{a\to b}$ represents a positive vote by user a on user b.

In decomposing the match probability into three parts, the first and third represent the likelihood that the voting user issues a positive vote. We represent in $p(vote^{+}_{a\to b}|vote_{a\to b})$ the likelihood that the user a will vote positive when we recommend user b. $p(vote_{b\to a}|vote^{+}_{a\to b})$ is the likelihood that user b will vote on user a following $vote^{+}_{a\to b}$, a probability that is itself influenced not only by the behavior of user b, say how active she is and how reliably she returns to Meet Me, but also by the implementation and rules of our algorithm, for example by how we rank user a among other users who have registered positive votes on user b, and by how often our recommendations of user b to others result in positive votes. The third component, $p(vote^{+}_{b\to a}|vote_{b\to a} \wedge vote^{+}_{a\to b})$ can be modeled similarly to the first component, for it represents the likelihood that a vote comes out as positive, yet since our application sometimes highlights match opportunities we do better by distinguishing this situation and training a separate model for it.

Ranking users b according to $p(match_{a\leftrightarrow b}|vote_{a\to b})$ is a reasonable first approach to the problems of making Meet Me recommendations for a. We will describe improved approaches later but first discuss early attempts at algorithm development.

2.2 Early Attempts

Our first algorithm implementations Meet Me were foundationally heuristic, only later coming to incorporate machine learning. We describe a progression of algorithms before outlining the challenges we encountered. These challenges arose not only from our approach to machine learning, but also from our system architecture, a traditional service-oriented web application, the layout and data flows of which are shown in Figure 2.

2.2.1 Heuristic Algorithms

An important early recommendation algorithm employed a patented approach [30] deriving inspiration from PageRank [29]. Ours may be the first commercial application of PageRank to social data, though Twitter also described a similar approach and popularized the notion of personalized PageRank in a social context [12]. Whereas the original PageRank algorithm for web search can be modeled as the likelihood of a page visit by a "random surfer" who starts at a randomly selected page, then traverses the graph along links between pages, at each hop continuing with probability α and stopping with probability $1 - \alpha$, the personalized PageRank algorithm starts the graph traversal at the node of interest, in this case at the user who is to receive recommendations.

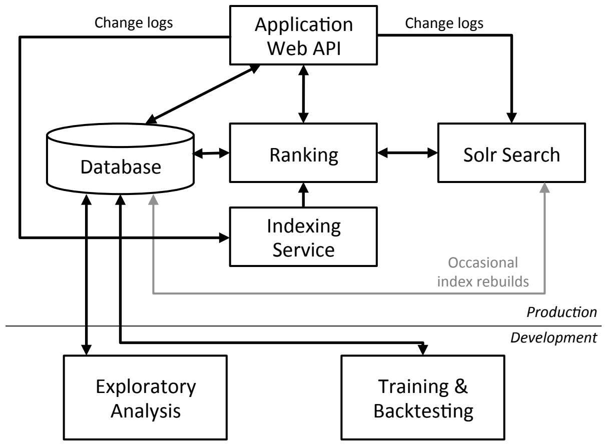


Figure 2: Architecture diagram of an early implementation of the Meet Me recommendation system. The API and database are based on standard web services technologies (PHP and Oracle). Recommendation candidates come from an Apache Solr search instance that first builds an index by querying the database, then stays current by processing change logs from the application. The ranking service (Java) operates similarly in maintaining an in-memory social graph, but also issues on-demand database queries to update user profile data. Data scientists engaged in development activities such as exploratory analysis, training, and backtesting query the database to extract working sets, most often studied using R and Python.

Our early work demonstrated the value of latent information present in social network interactions. For example, even without explicit data on age, gender, or sexual orientation, inspection of top results from a personalized PageRank query on the graph of friend connections or message exchanges gives results that are immediately recognizable as relevant (viewing a grid of photos can be a surprisingly effective way to get a quick and powerful impression of what an algorithm is doing, often proving more useful than statistical measures).

While our approach remained entirely heuristic, involving neither machine learning nor statistics, it provided plenty of parameters for experimentation. We focused on tuning parameters of the personalized PageRank algorithm, as well as parameters involving the level of user activity and the level of inbound positive interest. Lacking a predictive model of user behavior, we proceeded by intuitively guided trial and error, using judgment and quick sequences of A/B tests to maximize the number of users who received matches each day.

2.2.2 *Machine Learning*

We continue to believe that heuristics are a good way to start building recommendation engines, they test our problem understanding and can lead to good user experiences even with simple implementations. However, limited back-testing ability drives excess need for production experiments, and as the number of parameters rises it becomes increasingly awkward to reason about how to tune them manually. When we saw gains from heuristic improvements plateau we began to incorporate machine learning techniques, pursuing a promise of scaling to greater model complexity.

We chose to implement an SVM-based classifier predicting $p(vote^{+}_{a\to b}|vote_{a\to b})$ from a broad range of user details, not only age, gender and location, but also behavioral measures reflecting activity in sending and receiving friend requests and messages. We also included Meet Me activity, profile characteristics such as photos, schools, profile completeness, time since registration, profile viewing behavior, number of profile views received, ethnicity, religion, languages, sexual orientation, relationship status, and expressed reason for meeting people. Our approach might roughly be summed up as using any readily available information as a model feature, a contrast to the deliberate design approach we would later take.

This combination of machine learning with heuristics led to some gains at first, but we again soon found progress faltering. It was particularly troubling that the time between each improvement increased while gains realized in each decreased. In attempting to introduce new features to reflect user behavior better we encountered substantial software engineering challenges, including months spent making changes across multiple services, not only the ranking component but also the web application and database.

Among challenges we identified were the following:

- **Long deployment cycles:** Any algorithm changes required writing a large amount of software: SQL to extract historical data, Java code in models, Java code for processing real-time updates, often PHP code and more SQL to change how data was collected. For live experiments we also needed to consider how new and old implementations would coexist in production.

- **Limited feature transformations:** For the most part our classifier relied on features already available in the application, or those readily queried from the production database. These features represented data useful for display or for business logic, not necessarily for predictions. We lacked a simple and well-defined path for introducing new features, one with less overhead, one requiring effort commensurate to the complexity of the new feature rather than to the complexity of the architecture.

- **Difficulty in generating training data:** The database powering the application might store the current value of a feature, but might not retain a complete change log or history. If we were lucky we had access to a previous snapshot, but such point-in-time images would not accurately reflect the data available for real-time recommendations (see Figure 3). If unlucky, we would need to make new snapshots and wait for training data to accumulate.

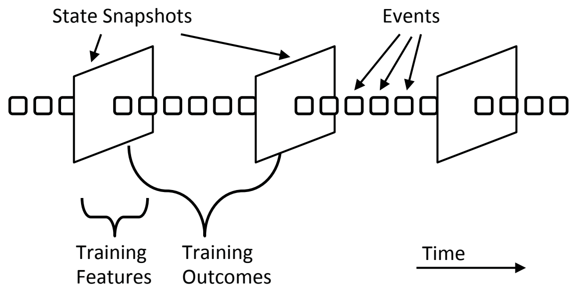


Figure 3: Training data consists of feature snapshots from the application database and outcomes occurring between them. These models are unable to capture feature variation between snapshots, and using real-time data in production introduces an inconsistency between model training and model deployment.

- **Lack of separation between domains:** We relied on computing features mostly in application code, creating a tight coupling between our recommendation system and our general business logic. We also mixed in-application feature computations with in-database computations expressed as SQL, furthering complex couplings.

- **Limited ability to backtest:** While we used training and cross-validation techniques in development of an SVM classifier, our recommendations remained dependent on a number of heuristic rules with tunable parameters. Our only path to tuning such parameters was through production experiments.

- **Limited problem insight:** *Ad hoc* data exploration and focus on statistical measures of performance left data scientists without a strong sense of user experience and, therefore, without the intuition necessary for breakthrough improvements.

- **Limited ability to collaborate:** We lacked a clear path to combine the efforts of multiple data scientists. We had only limited ability to deploy concurrent experiments, and the cost and complexity of implementing new features strained engineering bandwidth.

With so many challenges, we were lucky to have production experiments providing a safety net, protecting us against regressions as we stumbled towards improved recommendation algorithms.

The early approach described here has many shortcomings that leave it far from state-of-the-art. That said, in comparing notes with others we have come to believe that many of the challenges we encountered are common in industry. Our hope is that the solutions we share below will be broadly useful to those who deploy machine learning for high impact, as well as to those who plan to do so.

2.3 The Event History Architecture and Agile Data Science

Our answer to the struggles of previous approaches involves a number of deliberate choices: a departure from our previous software architecture and data architecture, specific ways of constructing machine learning models, and an adherence to certain ways of working with data—and with our team. These choices reinforce one another and allow an agile and iterative approach to delivering real-time recommendations for Meet Me.

2.3.1 Data and Software Architecture

Our architecture is driven by requirements, which can be summarized as follows:

- **Allow rapid experiments:** We should be able to go from ideas to validated results within two weeks.
- **Update in real-time:** Since user sessions are short, ~90s on average, it's important that models update within a few seconds, preferably <1s.
- **Support consumer internet scale and responsiveness:** Millions of active users, thousands of updates per second, subsecond response times for recommendations.
- **Encourage collaborative development practices:** Teams of data scientists and software engineers should make collective progress towards better Meet Me recommendations.

The architecture of our solution drops dependence on the relational database powering the web application, instead relying entirely on an event history log to support our machine learning systems. Importantly, we provide high-speed in-order access to event logs, and allow consumers to access both historical events and real-time streaming events using a single interface (see Listing 1). This architecture, diagrammed in Figure 5, puts the event history repository at the heart of the recommendation application.

It now becomes very natural to generate training and backtesting data for supervised learning algorithms. Whenever we encounter an event that we might want to predict, say $p(vote^+_{a\to b}|vote_{a\to b})$, we first write training data comprising the state of the model features just prior to the event occurrence, together with the event outcome. Only then do we update the state to reflect the event occurrence and continue rolling forward in time. Figure 4 illustrates generating training events in this way. The training data can be formatted for use with common statistical software, in our case R and Matlab.

Surprisingly, certain sorts of information that may not appear as event-like can benefit from representation in an event format. Take for instance zip code boundaries, ip-to-geography mappings, ISO country codes, or most any reference information that might naturally be implemented using a static lookup table. In many of these cases information can evolve, if slowly. By structuring such information as *fact update events* we maintain valuable flexibility and uniformity in our abstractions.

Another important benefit of the event history architecture is the symmetry it creates between historical backtesting and real-time streaming. We use the same feature definitions and state management software in development as we do in production. This proves key to quick deployments and rapid iterative data science cycles.

The event history architecture makes it practical to generate detailed training data for newly devised features, makes it straightforward and practical to deploy models based on such new features in production experiments, and consolidates features in one piece of code that works in both development and production. It provides composable abstractions that allow complex feature definitions, and it provides a single source of truth.

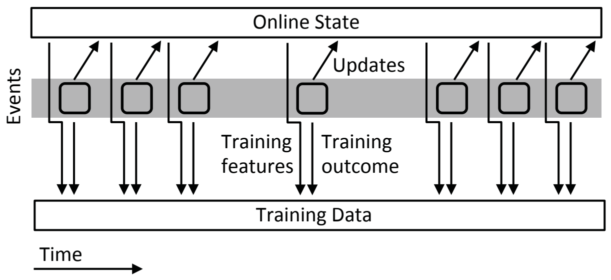


Figure 4: Training data generated from event history has granular alignment of feature state and training outcomes.

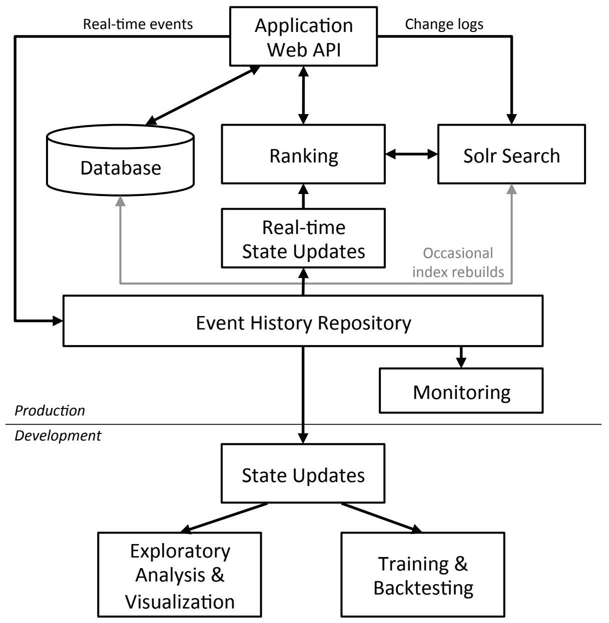


Figure 5: Diagram of the event history architecture. The event history repository serves as a central source of truth for production and development, and supports both historical access and real-time streaming.

2.3.2 Data Science

A central promise of Agile methodology is a more responsive development cycle, one that generates quick feedback through design, implementation, and validation phases, one that allows continuously incorporating learnings, making corrections, and exploiting opportunities. Another central promise is improving collaboration among team members. All of these are characteristics appealing for data science and for development of machine learning systems, and our approach delivers them in this context.

Figure 6 illustrates our cycle of iterative progress. It starts with data scientists developing problem understanding, and importantly, intuition. From here we propose model im-

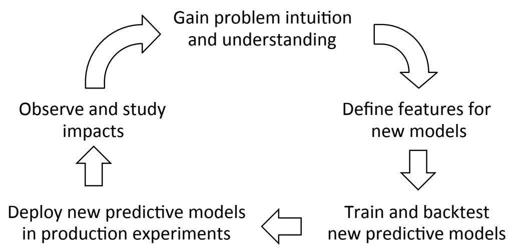

Gain problem intuition and understanding
Define features for new models
Train and backtest new predictive models
Deploy new predictive models in production experiments
Observe and study impacts

Figure 6: Agile data science cycle.

provements, typically in the form of new features. After training and establishing statistical basis for improvements through backtesting we deploy models to production and study impacts. We often realize gains but always improve our understanding, and enter the next development cycle with even stronger ideas. Additional details follow:

Problem intuition and understanding: Exploratory analysis often proceeds by asking simple questions, generally addressed with descriptive statistics and perhaps simple aggregates. We also encourage visualization, especially of user experience, and often find we can learn more from viewing just 100 faces, sampled from among 100,000, than we can from statistics on the aggregate. Working in this way not only helps us build our intuition but also our user empathy.

Models and feature engineering: We emphasize creative features with otherwise straightforward models. The event history architecture provides us with flexible feature design capabilities, and logistic regression serves us well, integrating signals typically represented by 50–100 features. Rather than learning a model with per-user parameters, we instead construct features that represent individual user characteristics, including how they relate to other users. These features evolve in response to individual user behavior, while the learned parameters of the model, applying to all users, remain fixed.

Training predictive models: Our event history accumulates several thousand events per second, and to compute features we need to process the entire stream. Early sampling is not an option because of the interconnected nature of our social network, and because features often capture interactions between users. On the other hand, since our models contain fewer than 100 parameters we are capable of generating much more training data than we need, at least over much of the domain of the feature space. This suggests that sampling is possible for training, however we are presented with another challenge: we must be careful not to introduce feedback, as occurs when the examples in the training set are biased because they are introduced by a recommendation algorithm. We address this problem by substituting a training recommendation, selected at random, in place of a ranked recommendation in 1% of instances. Doing so proved key to our ability to consistently achieve nonnegative performance changes when retraining models with more recent data, a foundational capability necessary for progress. We can imagine using more sophisticated sampling techniques to generate training examples, but benefiting from an enormous wealth of data we have not yet done so.

Production experiments: Our approach to large scale experiments is similar to that used by other consumer internet companies [18]. We encounter some special needs when testing Meet Me algorithms because performance ultimately depends on interactions between users. To address these, we developed a *split world* approach in which users are assigned at random to one of two partitions, seeing only users with the same "world" assignment in recommendations. Split world experiments are expensive because a reduced candidate pool degrades user experience, and because this drives us to limit ourselves to running one such experiment at a time. While we rely on split world experiments for final model acceptance, we start out evaluating model performance without a split world, assigning an initially small but progressively larger set of randomly selected users to the test group. During experiments we also take care to observe not only the initial effect, but also whether it grows or diminishes as user behavior adapts over time.

Production operations: A number of operating procedures support our Agile data science approach. We continuously monitor model performance, with loss and bias serving as indicators of problems. These measures can be essential for catching bugs that would otherwise go undiscovered amid the complexity of personalized recommendations.

2.4 Productivity and Business Results

Following our implementation of the event history architecture, our replacement of the original heuristic recommendation engine with a fully trained machine learning model implementation, and our adoption of a rapid-cycle Agile data science practice, we readily realized gains against our core optimization objectives. While the progress shown in Figure 7(a) represents contributions from various tuning efforts, including visual layout changes and promotions, results of experiments credit improved machine learning algorithms for over 30% increase in Meet Me usage.

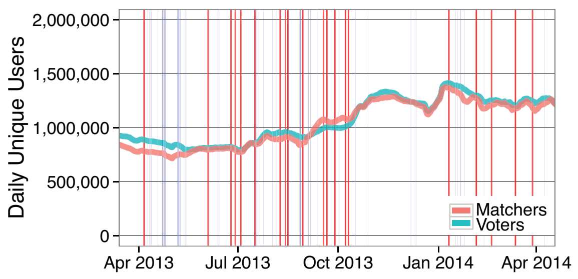
During the course of 12 months our team released 21 changes to the model and adjusted experiment parameters 163 times. During an especially intense 6-month period (May through October 2013) we released 15 changes to our models and adjusted experiments 123 times. We credit our progress to this rapid iteration. Unfortunately, towards the end of this period we suffered from increases in spam abuse (Figure 7(c)), forcing a diversion of attention from improving recommendations to addressing this ever-present threat. We have deployed a number of the techniques developed here in our latest anti-spam measures, but the topic is beyond the scope of this paper.

2.5 The Meet Me Implementation

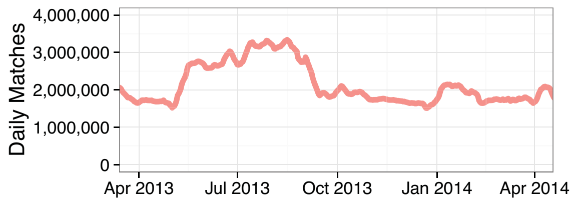
Here we attempt to provide a flavor for the models we have tested, though the full details of our Meet Me recommendation algorithm remain proprietary.

Optimization objectives: Perhaps the most straightforward approach is to optimize for the total number of matches, yet it quickly becomes clear that additional objectives call for consideration:

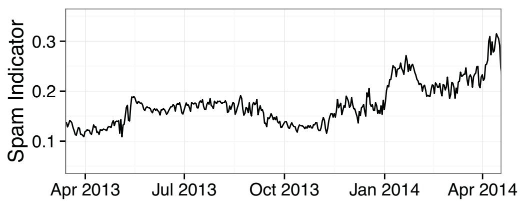
- *Total number of matches:* Simple to model by optimizing Equation 1, but gives too much exposure to active users with a high positive vote rate and fails to produce a good experience for many of the users.
- *Total number of individuals experiencing a match each day:* As Equation 1 but only the user's first match of the day counts towards the objective. This requires additional estimation but improves effectiveness.



(a) Meet Me daily unique matchers and voters.



(b) Meet Me daily matches.



(c) Meet Me spam index.

Figure 7: Meet Me metrics through a period of focused tuning. (a) shows progress towards increasing activity, with the 7-day average of daily voters and matchers overlaid on production changes. Vertical red lines indicate new algorithm releases, and blue lines indicate adjustments to experiment weights. (b) shows the 7-day average of daily matches, which doesn't correlate with other measures of user activity. Initial models designed to optimize matches led to limited increases in matchers and voters, whereas our later algorithms increased these metrics despite producing fewer total matches. (c) shows an index of spam, here provided by proxy of the female positive vote rate. Spam can contribute to inflated Meet Me metrics, especially matches, but this data indicates stable spam levels through the period of greatest gains. Near the end of the period our efforts shifted to combating increasing spam and away from improving recommendations.

- *Number of conversations following from a match:* We can consider the match successful only after message exchanges, or some other indication of a connection between people of a certain depth. Again, more modeling is required but we can hope for improved effectiveness.
- *Number of users engaged with the Meet Me feature on a daily basis:* Here we model how a recommendation impacts the probability that users involved will come back to the product on a future day. Reasons for re-engaging could include responding to a match opportunity, conversing following a match, or reacting to the present experience, perhaps with encouragement, perhaps with renewed determination to achieve a match.

While we have experimented with all of the approaches listed above, we achieve best results with the last alternative, by optimizing for the number of users predicted to engage with Meet Me on future days.

Models: We separately model the likelihood of a positive vote, either $p(vote^+_{a\to b}|vote_{a\to b})$ or $p(vote^+_{b\to a}|vote_{b\to a} \wedge vote^+_{a\to b})$, and the likelihood of a user returning to vote $p(vote_{b\to a}|vote^+_{a\to b})$. For the positive vote likelihood we use logistic regression, for one because supporting tools are well-developed, but also because it provides a probability as output. This allows us to verify model calibration, and provides a well-defined interface between separate model components. We have also incorporated a decision stump model, but continue to use logistic regression to calibrate it to outcome probabilities.

We estimate time to return using the exponential-response variant of a generalized linear model, with a threshold time to obtain a return vote probability. We note that this approach is a simplified variant of the hazard based approach to user return time prediction developed by Pandora [17].

Features: We use online features only, requiring quick updates for new events and efficient in-memory implementations. Features must also be very quick to access during production ranking, allowing for only a few memory accesses and perhaps some simple arithmetic, e.g., computing a ratio. A good feature is stationary, meaning that with consistent user behavior it asymptotically approaches a fixed value. For example, the number of votes in the past week represents a stationary feature, as does the fraction of all votes that are positive, whereas the total number of votes or the total number of positive votes do not represent stationary features. Some of the features we have implemented include the following, listed along with examples:

- *Binary profile indicator:* Is this user in the US?
- *Factor indicators:* Group all countries into 20 buckets, then provide a binary indicator of user location for each.
- *Combination features:* Product of factor indicators of two users, say (country bucket of user a) $\times$ (country bucket of user b).
- *Arithmetic features:* Age difference, square of age difference, difference of square of age, or any other function of the ages of two users.
- *Regular expression features:* Does the user send messages matching a certain spam indicator?
- *Ratio features:* Fraction of past votes by the user that are positive.
- *Filtered features:* Fraction of past votes by the user that are positive, calculated for each of various recommended user country buckets.
- *Threshold features:* Binary indicator of whether some measure, perhaps a ratio, exceeds a threshold value.
- *Most recent value features:* When was the user most recently active?
- *Transformations relative to current time:* How long ago was the user most recently active?
- *Exponentially smoothed features:* What is the exponentially smoothed voting rate $\sum_i e^{-k(t-t_i)}$ for votes at times t_i, current time t and smoothing constant k?

Our approach stresses the composability of models and features. We routinely rank results according to the combination of several models, and we have experimented with unsupervised models that feed into features of higher-level supervised models. Further details of the features described above are available as part of the Antelope open source software described in Section 3.

2.6 Software Engineering Notes

While if(we) eschews the software development culture of "not invented here", instead making extensive use of both commercial and free software packages, the Meet Me recommendation system consists almost entirely of custom code. Our abstractions and trade-off choices are somewhat different from those used in batch systems such as Hadoop, from those used in relational databases, and from those used by other stream processing software such as Spark Streaming, different enough that we choose to develop our own implementations using Scala and Java.

The event history repository is a service implementing an interface similar to that outlined in Listing 1. This interface highlights the essential character of the event history. It can only do two things: 1. receive and store new events, 2. return events in time-order, possibly applying some filter. By specifying an *endTime* in the future (typically $+\infty$) the client application gains access to real-time streaming, and in a single call to *getEvents* can access both historical and future events.

```
trait EventHistory {
  def publishEvent(e: Event)
  def getEvents(startTime: Date, endTime: Date,
    eventFilter: EventFilter, eventHandler: EventHandler)
}
```

Listing 1: The EventHistory interface (simplified).

We maintain feature state using large arrays of primitives, a packed in-memory representation allowing us to support over 10,000,000 candidate user recommendations on a commodity server. Our framework takes care of mapping feature state to array indexes. Our present implementation supports only a dense feature layout but we can imagine implementing a layout supporting sparse feature vectors. We spent some time considering whether to lay out our features row-major or column-major format, opting for the row-major format so as to keep our implementation simpler.

Our implementation is an efficient one that seeks to limit object allocation and that aims to provide good optimization opportunities to the JVM. While we can rank >50,000 candidate users per second, we see opportunities for order-of-magnitude improvements through optimization.

High availability and scalability are provided by replica ranking servers, while a single feature building server propagates state changes. In the event of feature builder failure we can restart processing from the event history log, accelerating recovery with saved snapshots. Even at our scale, we choose to optimize for performance rather than introduce distributed system complexities. A modern commodity server can handle >10,000 updates per second across 100 or more features, leaving us with ample headroom. We believe that with such implementations even many large businesses have no need for distributed architectures.

3. THE ANTELOPE REAL-TIME EVENTS FRAMEWORK

The event history architecture, our approach to feature engineering, and our approach to Agile data science are general, however the implementation for Meet Me recommendations remains coupled both to the problem and to the if(we) platform. The Antelope open source project[1] aims to make our approach broadly available. It presently represents a demonstration rather than a production tool, yet it fully illustrates the flexibility of the event history architecture, provides concrete examples of feature engineering, and serves as a guide for other implementations.

In our first example application, we address a Kaggle challenge for product search and recommendations [1]. Using the *Feature* interface of Listing 2 in the context of learning to rank [14], we demonstrate use of a simple *Popularity* feature along with a more complex *TfIdfFeature*.

```
trait Feature {
   def score(ctx: ScoringContext): Long => Double
}

class Popularity extends Feature {
  val ct = counter(skuViewed)
  override def score(implicit ctx: ScoringContext) = {
    id => ct(id) div ct()
  }
}

class TfIdfFeature extends Feature {
  val terms = counter(skuUpdated,
    productNameUpdatedTerms)
  val docsWithTerm = set(productNameUpdatedTerms,
    skuUpdated)
  val docs = set(skuUpdated)
  override def score(implicit ctx: ScoringContext) = {
    val queryTerms = ctx.query.normalize.split(" ")
    val n = docs.size()
    id => (queryTerms map { t =>
      val tf = terms(id, t)
      val df = docsWithTerm.size(t)
      sqrt(tf) * sq(1D + log(n / (1D + df)))
    }).sum
  }
}
```

Listing 2: The Feature interface and example implementations (simplified).

In our second example, we simulate user activity in an online dating context, then build recommendations using a model with features similar to those we have deployed for Meet Me (described in Section 2.5).

4. RELATED WORK

We present our review of related work in four parts: Agile data science, design and architecture of machine learning systems, choice of machine learning algorithms, and other approaches to real-time processing. While we find much of this literature to be complementary to the event history architecture described here, we find no direct precedent for our design.

Agile data science: While the Agile methodology [5] is well understood and broadly practiced in many variants by software engineers, the approach is only starting to make impact on data science. Jurney [15] provides a how-to guide for performing analytical tasks using Hadoop, encouraging adoption of Agile values. This work focuses on business understanding and does not address production recommenda-

[1]`https://github.com/ifwe/antelope`

tion engines, as we do. Beyond this, the dearth of literature on Agile data science is remarkable; we imagine this may change in coming years.

Machine learning systems: The work described here has strong philosophical parallels to the Hazy project [22], which states the goal of "making it easier to build and maintain big-data analytics." The emphasis on feature engineering, and the belief that simple machine learning models with more or better features will often outperform more sophisticated algorithms, rings true to our experience. We too emphasize programming abstractions and infrastructure abstractions, but choose rather different implementations. Whereas the Hazy project uses probabilistic logic programming, we use reactive-style Scala code to implement features. Our central infrastructure abstraction is the event history repository, whereas Hazy works against a static data model.

The Hazy team has also described a vision of a data system for feature engineering [4]. In asserting that "feature man-months aren't mythical," they suggest that whereas on traditional software projects adding more people rarely leads to proportionately faster progress [7], in trained systems loose coupling can permit a large team to work toward a common goal with only limited coordination. The proposal, most recently implemented in the DeepDive system [32], facilitates feature engineering as we do, yet does not adopt a time-based data model or provide for real-time updates.

Recent work by the UC Berkeley AMP Lab reveals Velox [10], a low-latency solution to model serving and management. Velox provides important operational capabilities not previously available in open-source machine learning toolkits. In addition to low-latency serving it provides an online update functionality that approximates continuous model retraining, as well as periodic batch retraining. We are not alone in recognizing the gap between the promise of machine learning applications and what is commonplace in industry. In comparison, by using online features only, we achieve continuous production operation without needing batch operations. Furthermore, in addressing feature engineering challenges we contribute additional productivity improvements.

There are a variety of toolkits designed to make machine learning more practical and accessible. MLbase [20] automates many of the technical steps required to build effective machine learning models. The venerable Weka [13] remains a powerful environment for machine learning, offering not only a large number of machine learning algorithms but also a variety of filters for preprocessing data. Our provisions for feature engineering are richer, and we can imagine complementing both of these toolkits.

Machine learning is applied extensively in online advertising, and the production techniques developed at Google are relevant to our work [27, 31]. While there is no mention of a central event history or of using event processing to support iterative development and expressive feature engineering, the importance of real-time processing, of a log-based approach, of keeping machine learning simple, and of thinking carefully about architecture all come through clearly.

Machine learning algorithms: We rank recommendations in Meet Me by ordering candidates according to the optimization objective, matches in early implementations, and predictions for overall engagement in later implementations. Here we review alternatives.

The learning to rank literature is perhaps best developed in the area of information retrieval [25], but its principles can apply to personalized people recommendations as well [14]. Our experience developing examples for Antelope suggests that learning to rank is a natural match with the event history architecture.

People recommendations struggle with the special challenge of a high-dimensionality target space; there are many more candidates than in e-commerce or in entertainment. Collaborative filtering has widely publicized success as part of Amazon's product recommendations [24], and related matrix factorization techniques were used in winning the Netflix prize [19]. While we did not pursue such approaches, it would be interesting to evaluate whether they are practical in our setting.

Contextual-bandit techniques have also been adopted for personalized recommendations [23], in an approach that we believe could be combined with the creative feature engineering described here. In particular, we are encouraged by results showing offline evaluation using previously recorded traffic, which dovetails with an event history approach.

Real-time processing: A number of architectural paradigms exist for processing real-time information.

Complex event processing is well established as an integration pattern that fits naturally with an Agile approach [9]. The MillWheel system described by Google [3] is a platform for reliable stream processing that can execute arbitrary imperative code. A log-centric architecture extolled by Kreps [21] is particularly well aligned with our perspectives on data processing; the immutable log represents the system's single source of truth, just like our event history. Similarly, Event Sourcing and CQRS (Command & Query Responsibility Segregation) [6] patterns for application architecture are particularly well suited to our approach. Streaming query engines are developed in the academic literature [8] and have commercial implementations [2]. All of these platforms and perspectives could serve as a part of an implementation for our architecture.

One approach to real-time recommendations is to build a speed layer on top of a batch system [26]. While this evolutionary approach is sensible, and while some calculations are easier with batch processing, we believe that in most cases such hybrid architecture introduces unnecessary complexity.

5. CONCLUSIONS AND FUTURE WORK

The event history architecture provides a powerful abstraction, allowing teams of data scientists to collaborate actively on machine learning projects. Our production implementation delivers personalized recommendations drawn from millions of candidates, answers queries in under a second and incorporates new information in seconds. We are presently working on Antelope, an open source implementation incorporating the event history paradigm and providing a flexible environment for feature engineering. We have used several machine learning tools alongside the technology developed here, and can envision deeper integrations, including ones with popular data management systems, which only need to implement the interface of Listing 1.

We remain struck by the gulf between sophisticated demonstrations of what is possible in machine learning, and the day-to-day realities of what is practical in most organizations. By bringing Agile capabilities to production data science applications we hope to narrow the gap, to help teams feel the thrill of frequently realized gains, and to help them build and deploy more of what they can imagine.

6. ACKNOWLEDGMENTS

This work reflects the industriousness and ability of the data scientists and engineers of the if(we) Relevance Team. Special thanks go to Karl Dawson and Helena Buhr for leadership and helpful discussions, and to Vinit Garg, Dai Li, Martin Linenweber, and Madhusudana Shashanka for tireless efforts.

7. REFERENCES

[1] Data mining hackathon on (20 mb) Best Buy mobile web site - ACM SF Bay Area Chapter. `http://bit.ly/1O3eDOD`. Accessed: 2015-02-20.

[2] Stream processing explained. `http://www.sqlstream.com/stream-processing/`. Accessed: 2015-02-20.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[4] M. Anderson, D. Antenucci, V. Bittorf, M. Burgess, M. J. Cafarella, A. Kumar, F. Niu, Y. Park, C. Ré, and C. Zhang. Brainwash: A data system for feature engineering. In *CIDR*, 2013.

[5] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, et al. The Agile manifesto. `http://agilemanifesto.org/`, 2001.

[6] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure.* Microsoft patterns & practices, 2013.

[7] F. P. Brooks Jr. *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition, 2/E.* Addison-Wesley Professional, 1995.

[8] S. Chandrasekaran and M. J. Franklin. Streaming queries over streaming data. In *VLDB*, pages 203–214, 2002.

[9] K. Chandy and W. Schulte. *Event Processing: Designing IT Systems for Agile Companies.* McGraw-Hill, Inc., New York, NY, USA, 2010.

[10] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. *CoRR*, abs/1409.3809, 2014.

[11] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, July 2013.

[12] P. Gupta, A. Goel, J. Lin, A. Sharma, D. Wang, and R. Zadeh. WTF: The who to follow service at Twitter. In *WWW*, pages 505–514, 2013.

[13] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The WEKA data mining software: An update. *SIGKDD Explor. Newsl.*, 11(1):10–18, Nov. 2009.

[14] L. Hong, R. Bekkerman, J. Adler, and B. D. Davison. Learning to rank social update streams. In *SIGIR*, pages 651–660, 2012.

[15] R. Jurney. *Agile Data Science: Building Data Analytics Applications with Hadoop.* O'Reilly Media, 2013.

[16] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2917–2926, 2012.

[17] K. Kapoor, M. Sun, J. Srivastava, and T. Ye. A hazard based approach to user return time prediction. In *KDD*, pages 1719–1728, 2014.

[18] R. Kohavi, A. Deng, B. Frasca, T. Walker, Y. Xu, and N. Pohlmann. Online controlled experiments at large scale. In *KDD*, pages 1168–1176, 2013.

[19] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[20] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. MLbase: A distributed machine-learning system. In *CIDR*, 2013.

[21] J. Kreps. The log: What every software engineer should know about real-time data's unifying abstraction. `http://linkd.in/1fDnlQk`, Dec. 16 2013.

[22] A. Kumar, F. Niu, and C. Ré. Hazy: Making it easier to build and maintain big-data analytics. *Commun. ACM*, 56(3):40–49, Mar. 2013.

[23] L. Li, W. Chu, J. Langford, and R. E. Schapire. A contextual-bandit approach to personalized news article recommendation. In *WWW*, pages 661–670, 2010.

[24] G. Linden, B. Smith, and J. York. Amazon.com recommendations: Item-to-item collaborative filtering. *Internet Computing, IEEE*, 7(1):76–80, 2003.

[25] T.-Y. Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, Mar. 2009.

[26] N. Marz and J. Warren. *Big Data: Principles and best practices of scalable realtime data systems.* Manning Publications Co., 2015.

[27] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, et al. Ad click prediction: A view from the trenches. In *KDD*, pages 1222–1230, 2013.

[28] B. Meyer. *Agile!: The Good, the Hype and the Ugly.* Springer Science & Business Media, 2014.

[29] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the web. 1999.

[30] J. Schleier-Smith. System and method of selecting a relevant user for introduction to a user in an online environment, June 17 2014. US Patent 8,756,163.

[31] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, and M. Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[32] C. Zhang, C. Ré, A. A. Sadeghian, Z. Shan, J. Shin, F. Wang, and S. Wu. Feature engineering for knowledge base construction. *CoRR*, abs/1407.6439, 2014.