ItemSuggest: A Data Management Platform for Machine Learned Ranking Services

Sandeep Tata, Vlad Panait, Suming J. Chen, Mike Colagrosso
Google
1600 Amphitheatre Parkway
Mountain View, CA, USA
{tata, vpanait, suming, mcolagrosso}@google.com

ABSTRACT

Machine Learning (ML) is a critical component of several novel applications and intelligent features in existing applications. Recent advances in deep learning have fundamentally advanced the state-of-the-art in several areas of research and made it easier to apply ML to a wide variety of problems. However, applied ML projects in industry, where the objective is to build and improve a *production* feature that uses ML, continues to be complicated and often bottlenecked by data management challenges.

ItemSuggest is a platform for building contextual relevance services. In this paper, we describe ItemSuggest with a focus on how we leverage key ideas from data management to make it dramatically easier to build machine-learned ranking services. The platform allows engineers to focus on application-specific modeling and simplifies key tasks of 1) gathering training data; 2) cleaning, validating, and monitoring data quality; 3) training and evaluating models; 4) managing the feature lifecycle; and 5) running A/B tests. We outline key design choices anchored around the core idea of optimizing for experiment velocity. We describe lessons learned from applications built on this platform that have been in production serving hundreds of millions of users for over a year. Finally, we identify two key components of the platform where data management research can have a major impact—a transformation engine for feature engineering and one for training set representation. We believe such platforms have the potential to accelerate and simplify ML applications the same way data warehouses radically simplified complex reporting applications.

1. INTRODUCTION

Recent advances in ML are opening up several new opportunities [10] in machine translation, smart email replies, better search [14], and document finding [21]. Most practitioners, however, continue to emphasize that 80%–90% of the time spent in an applied ML project is in data management [12, 13, 17]. Lack of effective data management techniques continues to be a key impediment. The infrastructure available to data scientists today is analogous to what was available to data analysts before modern data warehouses and ETL solutions radically improved their productivity.

This paper motivates and presents the design of ItemSuggest: a data management platform for deploying machine-learned ranking services. Ranking a set of choices in a given context is a key abstraction that fits a large number of practical ML problems in search (ranking given an explicit query) and recommendations (ranking a set of candidates without an explicit query). The platform provides a clean abstraction for complex, error-prone, data-management tasks such as data collection, augmentation, cleaning, transformation, online experiments, and feature lifecycle management. ItemSuggest allows the data scientist to focus on modeling and experimentation. As of the writing of this paper, ItemSuggest is being used for approximately ten different ML applications, including Quick Access and zero-prefix search suggestions for Google Drive.

This paper focuses on two key contributions: First, we advocate for a common server architecture where multiple teams build and deploy application-specific code for candidate-generation, feature computation, and label acquisition. Second, we advocate (perhaps controversially) for gathering training data online rather than relying on data dumps from other sources for production models. The common server architecture provides a natural place to implement a consistent data acquisition pipeline.

The common server is a key component of the platform, and is roughly analogous to how application servers provided a clean framework to manage business logic separately from scaling and performance administration concerns. Application servers were successful in accelerating the pace at which enterprise and web applications have been deployed. We believe that this architecture is likely to have the same impact for ML ranking applications. As discussed later, the ItemSuggest server standardizes on a common schema, the TensorFlow Example [1] protocol buffer, to represent training data.

Most ML projects focus on getting data dumps from existing sources rather than implementing an end-to-end logging and data acquisition pipeline. While this is a reasonable way to prototype an initial model, we advocate that a production model should have a pipeline to instrument and gather its own training data. This is based on our experience building Quick Access [21], a service that suggests the documents that a user is likely to open when she arrives at the home screen of Google Drive. Quick Access was built from scratch,

9th Biennial Conference on Innovative Data Systems Research (CIDR '19) January 13-16, 2019 , Asilomar, California, USA.

and then rebuilt on ItemSuggest. A major setback during the first deployment of Quick Access was the discovery of *train-serve* skew, where the features supplied to the model at the time of serving (inference) turned out to be significantly different from what was available in the data dumps used for training [21]. Analysis showed that the skew originated from two separate causes: First, the service that supplied features to the model at inference-time applied several transformations on the data available through the dump before responding to requests. This was desirable for other clients of this service, but particularly bad for an ML application. The second cause was from data propagation delays. Data from the dump contained all events up to the instant of the scenario for which we were trying to construct an example. In contrast, data from the service was often missing the last few minutes of activity. In retrospect, this wasn't surprising given that data logged in various data-centers around the world needs to be processed before being made available through this service. What was surprising was the impact of this delay. The train-serve skew wiped out nearly all of the advantage the ML model had in offline experiments over a simple heuristic baseline. Please see Section 4.2 in [21] for a more detailed explanation.

The risks of this happening are greater when the team in charge of the data infrastructure is different from the team responsible for training and evaluating a suitable ML model. The risks are also significantly higher when the training data is constructed by joining data from multiple sources, each managed by different teams, with varying levels of documentation. In the case of Quick Access the current service queries seven different data services, each managed by a separate team, to gather the data required to apply the ML model.

In addition to being able to avoid a complex offline join across multiple data sources, the training data thus collected avoids skew from undocumented transformations, as well as pipeline delivery delays which might not be apparent in a data dump. Applications built on the ItemSuggest service follow a *dark-launch and iterate* approach where an initial model is launched to just gather the necessary training data without affecting any production traffic. Our experience so far suggests that applications are built significantly faster and with far fewer data quality issues on ItemSuggest than otherwise.

The rest of this paper is organized as follows. Sections 2 and 3 describe the common server architecture and the accompanying data infrastructure respectively. Section 4 describes a few applications that have been built on the ItemSuggest platform. Section 5 summarizes related work, and Section 6 points to potential areas for future data management research in this space.

2. SYSTEM ARCHITECTURE

In this section we give an overview of ItemSuggest architecture. We discuss this from the perspective of a developer wishing to build an application on ItemSuggest, using Quick Access as a running example. ItemSuggest provides two APIs that a client can call. An application developer needs to provide a concrete implementation for both of these:

- *ListSuggestions*—to request a list of ranked suggestions for an application. The results depend on the *ApplicationSettings* (described in the next section) the developer has configured and registered with the ItemSuggest server.
- *RecordFeedback*—to provide a record of user feedback (e.g. clicks) on the list of suggestions back to ItemSuggest.

2.1 ListSuggestions

The ListSuggestions API is what the application developer must implement in order to generate suggestions for the end user. The developer first configures the *ApplicationSettings* for their application. These include the the set of candidates to consider, additional sources of data that are necessary, and the ML models to call for ranking, etc. For Quick Access, a developer would need to configure a ListSuggestions call to retrieve a list of relevant documents for a user. The ListSuggestions call contains:

- Client-side information that is not available on the server-side. An example would be the device-type that is issuing the request (mobile phone vs. browser on a laptop).
- A desired output format for the returned suggestions. This specifies the number of results and the details required for each suggestion.

A more detailed breakdown of ItemSuggest's architecture can be seen in Figure 1, where the ListSuggestions call is processed in four stages to finally produce the output—a ranked list of candidates.

The developer configures each of these stages using *plugins* that may be reused across applications. The *plugins* are the main business-logic building blocks. The configuration for each plugin and the execution graph (how the plugins depend on one another) is provided via the *ApplicationSettings*. Plugins allow for sharing and reuse of infrastructure and sources. For example two very distinct intelligence products can reuse the same plugin to get information about upcoming Calendar meetings. This can be achieved with a trivial update of the corresponding *ApplicationSettings*. Plugins also allow uniform processing, checks and best practices. The objects constituting inputs and outputs to each of plugins are protocol buffers [22].

2.1.1 Context Creation

There are multiple data-sources that have been set-up for use within ItemSuggest, ranging from users' Drive Activity to their Calendar events. A developer can specify in ApplicationSettings which set of existing sources should be utilized (via the plugins configured for execution). In this stage, multiple RPCs are scheduled in parallel to each of the data-sources to retrieve and combine the results to create the user *Context*. The Context can be seen as the general information available about a particular user at the time of request.

$ContextCreation(ListSuggestions) \rightarrow Context$

2.1.2 Candidate Creation

In this stage, Candidates are generated from the previous stage's Context. Note that "candidates" is very loosely defined, and in ItemSuggest candidates can be anything from documents, emails, meetings, people, to queries. For Quick Access, application code is written to specify how candidates should be built from the Context. For instance, we combine

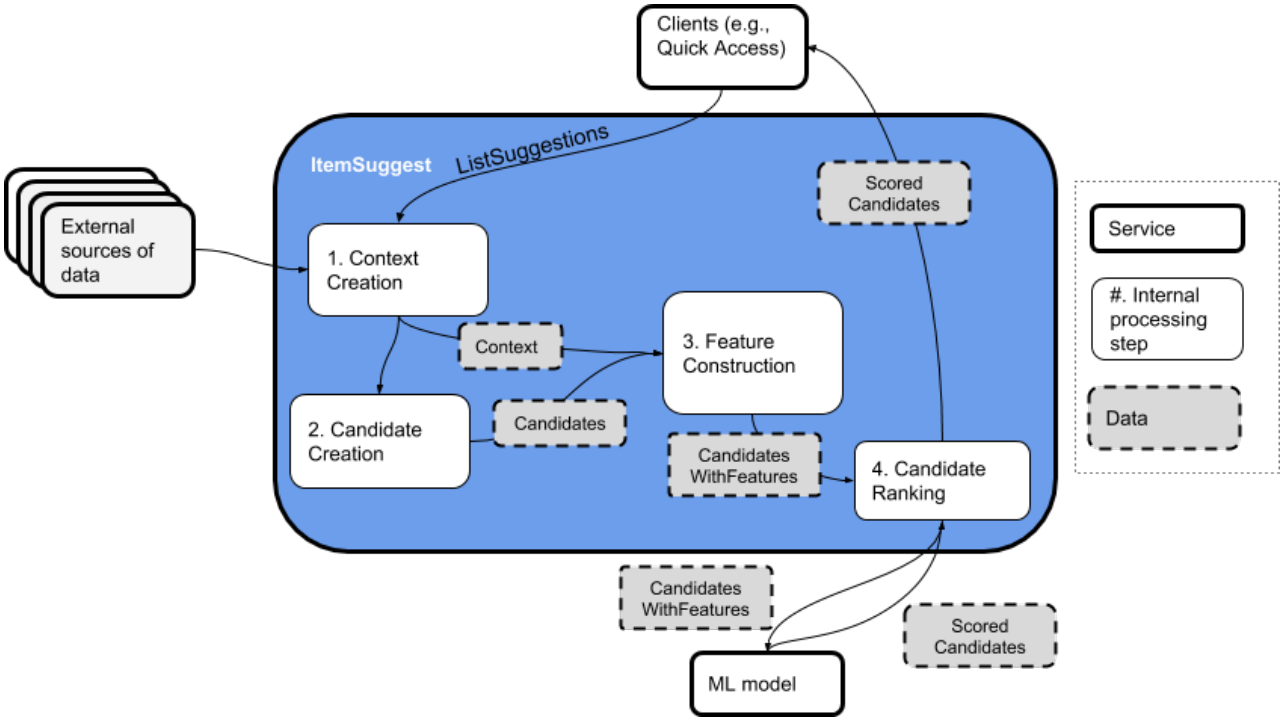

Figure 1: ItemSuggest architecture for generating a list of suggestions.

candidates from two sources—a service that reports documents with recent activity, and another that returns all the documents in a user's root folder to construct candidates for Quick Access. It is the application developer's responsibility to configure *what* sources of data should be in the Context and *how* Candidates should be generated from various sources.

$CandidateCreation(Context) \rightarrow Candidates$

2.1.3 Feature Construction

The input to this stage are Candidates that have been generated along with the Context. Additional features can be extracted for the candidates using the Context (e.g. what time the candidate was most recently accessed, how many times the candidate was opened). Further, some candidates can be excluded based on product constraints. For instance, an application like Quick Access may choose to exclude image-types from being considered for suggestion.

$FeatureConstruction(Candidates) \rightarrow CandidatesWithFeatures$

2.1.4 Candidate Ranking

The input to this stage is an unsorted list of Candidates. The output is a list of Candidates that have been ranked by either a machine-learned model or a heuristic. For new products application developers generally start with simple heuristic rankers that are easy to interpret and provide reasonable results. After the application is dark-launched, training data can be gathered to build machine-learned models. The default ML ranking strategy in ItemSuggest makes a blocking RPC call to a service (like TensorFlow Serving [18]) that scores the candidates. The scored candidates can then be sorted to return the top n suggestions.

$CandidateRanking(CandidatesWithFeatures) \rightarrow ScoredCandidates$

2.2 RecordFeedback

The RecordFeedback API call allows the the application developer to record any feedback from the presented list of suggestions. The recorded feedback is sent back to ItemSuggest. A typical example is user interaction information such as which of the suggestions (if any) were clicked. A logging service is responsible for collecting the data from RecordFeedback calls and joining them with the data logged by the corresponding ListSuggestions call. This data is then used to produce training data and compute metrics as described in the next section.

3. DATA INFRASTRUCTURE

3.1 Data Acquisition

Figure 2 shows how ItemSuggest processes RecordFeedback calls to gather training data. ItemSuggest logs the results of every ListSuggestions call and RecordFeedback call to a logging service. This service is responsible for making available a periodic joined result that pairs the candidates and their features generated by a ListSuggestions call with the results returned by an eventual RecordFeedback call. The join is facilitated by generating a unique ID and including it in the response to each ListSuggestions call. Clients then pass back the ID when calling RecordFeedback. Note that a RecordFeedback call may arrive at an arbitrary point in time after the ListSuggestions call and matching the RecordFeedback call with the corresponding ListSuggestions call in the ItemSuggest server before logging to storage was

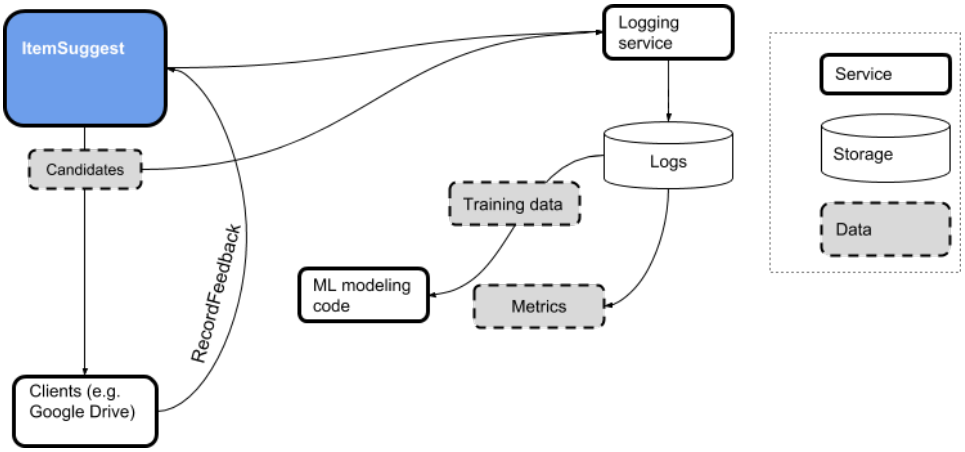

Figure 2: Architecture of ItemSuggest's integration with machine-learning workflow.

not a practical design. The joined data serves as the input to various parts of the ML workflow. In order to generate training data for an ML model, we only need to implement a transformation from the results of ListSuggestions calls and RecordFeedback calls to a set of (tf.Example) with labels. For readers not familiar with the tf.Example protocol buffer, for the purposes of this discussion, one may think of it as a map of named *Features*, where each Feature is a list of either integers, floats, or byte-arrays. Training infrastructure is detailed further in Section 3.4.

A typical non-ItemSuggest project might use existing client application logs and join them with reference data available on the server-side to compute features and examples. Instead, with this approach, we have a common data acquisition platform that manages this often complicated join across all the ItemSuggest applications. Furthermore, this approach dramatically simplifies the need for data cleaning and monitoring for train-serve skew. The one downside is that in order to start training a model, developers need to first deploy an application in ItemSuggest with a heuristic (or dummy) ranker to gather training data. You may need to gather several days worth of traffic before training an initial model. In practice, we have found that the data cleanliness provided by this approach far outweighs the additional velocity one may get from using data dumps to train models.

3.2 Feature Construction

The feature construction module is a data transformation engine that accepts context data along with candidates, and outputs a set of tf.Examples that can be sent to the ranker for scoring. This is the first of two data transformation components we describe. We start with the example of Quick Access to illustrate the computation in this module and argue why organizing it efficiently is important. A key data source for Quick Access is the Activity Service in Drive. It gathers all requests made to the Drive backend by any of the clients (web, desktop, Android and iOS clients, and third-party apps). It logs high-level user actions on documents such as create, open, edit, delete, rename, comment, and upload events.

To describe how features are constructed for each candidate, we introduce two abstractions: the *FeatureGenerator* and the *ExampleGenerator*. The FeatureGenerator accepts an instance of *Context* and optionally, a Candidate and produces a tf.Feature. FeatureGenerators are also explicitly associated with one or more external data sources so the computation can be scheduled as soon as the RPC to its data source has returned. An ExampleGenerator is configured as a map of feature names to *FeatureGenerators*. For each ListSuggestions call, the ExampleGenerator is invoked with the Context and batch of Candidates to produce a batch of tf.Examples that are sent for scoring. The API for implementing a *FeatureGenerator* allows developers to build up common data structures that are reused for each candidate. For example, "TimesGenerator(WEB, OPEN)" constructs a FeatureGenerator that produces a vector of recent open events for a given candidate document on the web. It represents the timestamps for these open events relative to the request time (available through the implicit context) and represents the values in seconds as a float vector of a given length, say l, using the l most recent events. If fewer events than l are found, the remaining values are padded with zero. FeatureGenerators vary in complexity—"current time" would be a trivial candidate agnostic feature, and "recency rank" is more complicated requiring computation spanning multiple candidates. The current Quick Access model uses several FeatureGenerators that in aggregate produce well over 10K floats for each example. A sample configuration for an ExampleGenerator is shown below:

```
FeatureGenerator f1 = new TimesGenerator(WEB, OPEN);
FeatureGenerator f2 = new TimesGenerator(MOBILE, EDIT);
...
FeatureGenerator mime = new MimeGenerator();
...
ExampleGenerator e = ExampleGeneratorFactory().
                       .addFeature("web-opens", f1)
                       .addFeature("mobile-edits", f2)
                       ...
                       .addFeature("mimetype", mime)
                       ...
                       .build();
Examples batch = e.compute(context, candidates);
```

This is a critical component with two competing goals—low latency and ease of experimentation. The logic for con-

verting the context and candidates to tf.Examples needs to be run in response to every single suggestion request. This puts the component in the critical path for the overall latency of the the suggestion service. The latency budget available to the machine-learned ranking component is often under 50 ms. This needs to be shared between feature computation and model evaluation. As a result efficient feature computation is vitally important. On the other hand, engineers tasked with improving the quality of a model constantly add experimental features and train new models that try to take advantage of these features. As a result, the feature construction module needs to be able to allow engineers to quickly develop new features in parallel and deploy them so they are available for training and experimentation.

The design of the feature construction module tries to accommodate these two goals. The module provides an API where engineers may add features independent of one another (with potentially redundant computation). At runtime, an optimized version of this code is run with parallelization to minimize latency. The API also allows developers to separate production vs. experimental features. Production features must be computed to serve a user request, while experimental features can be computed asynchronously and logged without adding to the overall latency. This is particularly valuable for evaluating computationally expensive features.

Feature construction is essentially a data transformation computation where a Context protocol buffer and a set of Candidate protocol buffers are transformed to a set of tf.Examples. Each FeatureGenerator can be viewed as a piece of the overall transformation, and the ExampleGenerator organizes this computation for a batch of Candidates, currently using a fixed plan. The best approach for designing such a component is an open research question. Is this data transformation best expressed in code or declaratively in a transformation language? Could a Domain Specific Language (DSL) open up opportunities for optimization while keeping this easy to read and maintain?

3.3 Feature Lifecycle Management

A major portion of an ML project's lifecycle involves adding new features, training new models, and experimenting with them to see if they improve a metric of interest. In ItemSuggest, an engineer follows a standard prescription for adding and removing features. She implements a new FeatureGenerator, registers it with a unique name, and updates the feature schema for the application. The feature schema is used to maintain a consistent view of what features are computed in response to each ListSuggestions call and the subset of features required for each model. The schema ensures that ItemSuggest doesn't accidentally invoke a model with only a subset of the features it expects, resulting in the model failing to provide any ranking scores.

Once a feature is implemented, ItemSuggest takes care of logging this new feature for future requests. Data that was logged before this feature was added is *not* modified to include this feature. When generating training data for a model that requires this feature, ItemSuggest automatically excludes older data that does not conform to the schema expected by this model.

There may be multiple engineers working concurrently on adding different sets of features or even removing features. Schemas provide a way to enforce that some new features

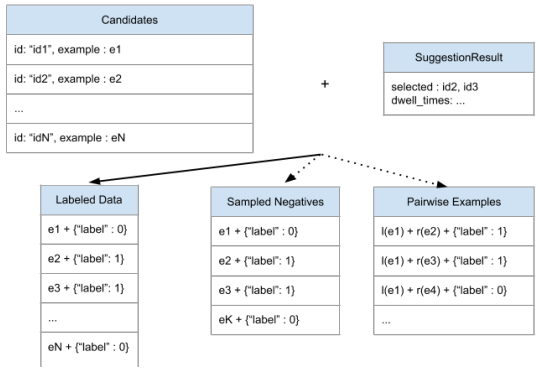

Figure 3: Generating labeled examples from Candidates and RecordFeedback

that are newly logged and in "experiment" mode won't interfere with another engineer's work on removing features.

Consider the following example that illustrates the use of schemas in managing the addition and removal of features. Assume that a given application uses a set of existing features denoted by E. Engineer A adds a new feature ($A1$) that she wants to include in a new model. Engineer B adds new features $B1$, $B2$ that she wants to experiment with. Engineers A and B may only have approval to log the experimental features for 10% of traffic until the features are deemed valuable enough to justify the additional CPU or storage cost imposed by the new features. Engineer A can create a new schema file containing $E \cup A1$. Engineer B may create one or more schemas $E \cup B1$, $E \cup B1, B2$ for each of the experiments she wants to run. When generating training data or computing the example representation to be scored by a model in a live experiment, ItemSuggest uses the schema files to ensure all required features are computed. ItemSuggest engineers also have the ability to deprecate a feature by adding it to an exclusion list that prevents any new schemas from utilizing that feature.

After training a new model (say with a new feature $A1$), starting an experiment is as easy as specifying in a config file a certain percent traffic that the model should serve. Since feature $A1$ had to have been logged by ItemSuggest in order to train this model, it already is available to be consumed for serving.

3.4 Training Infrastructure

The infrastructure is designed with the primary goal of making the core idea–implement–train–evaluate loop of a data scientist seamless and fast. As mentioned in Section 3.1, the logging service makes available the joined data consisting of the Candidates considered for each ListSuggestions call (along with the Features computed for them) and the results from the RecordFeedback call. In order to generate labeled examples, one only needs to implement a transformation from (Candidates, RecordFeedback) to tf.Examples. A simple version is pictorially depicted in Figure 3 in the flow producing "Labeled Data."

Note that while the transformation from source protocol buffers to tf.Examples appears similar to that described in Section 3.2 on feature construction, this does not execute in response to a user request. Instead, this transformation code should be designed to support experiment velocity rather

than transformation latency. Data scientists empirically try different ways of representing the training data. One example is to sample the negatives if we have far more negative examples than positives for some users (depicted using "Sampled Negatives" in Figure 3). We may consider and score hundreds of Candidates in response to the ListSuggestions call, display a few, and see that the user chose only one Candidate. Instead of producing training data with one positive and hundreds of negative examples for this case, we may want to downsample the negatives (the Candidates that were not clicked). Another modeling approach is to use pairwise examples (depicted as "Pairwise Examples" in Figure 3) to represent the fact that one candidate was preferred over another. While more complex, these have been shown to outperform point-wise models [7]. Another modeling idea would be to weigh clicks from a certain UI sections or platforms more than others (mobile clients tend to have more accidental clicks than web clients because of small screens).

A straightforward solution to this transformation problem would be to implement this in a Flume [8] job and materialize the results to be picked up by a training job in TensorFlow. This was indeed our initial solution. Our current approach allows us to represent this transformation concisely in C++ code and run this either in a Flume job or directly in the TensorFlow graph. This allows us to represent the training data set as a view over the underlying pair of (Candidates, RecordFeedback) so an engineer may try out many ideas quickly without having to first materialize large data sets.

Most ItemSuggest applications use a canned neural network of feed-forward layers that is known to be robust. TensorFlow supports more than just neural networks, and applications are free to choose other models like Random Forests or generalized linear regression.

3.5 Framework for Metrics and Experiments

The common server architecture and log format allows ItemSuggest to provide several turnkey metrics like Clickthrough Rate (CTR), Mean Reciprocal Rank (MRR), and Average Click Position (ACP) to all applications. These are common metrics for ranking problems and generally useful across applications. ItemSuggest supports A/B experimentation for both percent of traffic as well as by certain groups of users. This experimentation can be done on a common server and requires no additional infrastructure.

4. APPLICATIONS

We measure the success of our platform by the velocity of production applications that we have built, launched, and optimized on it. Because of the common architecture and shared data infrastructure, our pace has accelerated, rather than slowed down, as we have added more applications.

After rebuilding Quick Access on ItemSuggest, we have increased the number of engineers that can concurrently try new features and train models. Our experience shows that we have not only been able to rapidly improve models for Quick Access, but new applications have taken significantly fewer resources—both in terms of number of engineers and the time taken to build.

Quick Access originally launched in a user's root folder, called My Drive. Quick Access now also makes suggestions for enterprise users who put their files in a shared Team Drive. Figure 4 shows Quick Access suggestions in a Team Drive on the left. In terms of ItemSuggest infrastructure, Quick Access for Team Drives uses the same Feature Selection and Candidate Ranking stages. The Context Creation and Candidate Creation phases are different because the context contains the Team Drive the user is viewing and the contents of the shared Team Drive comprise the candidates created. Because of the shared stages, Quick Access for Team Drives also improves when the base model improves.

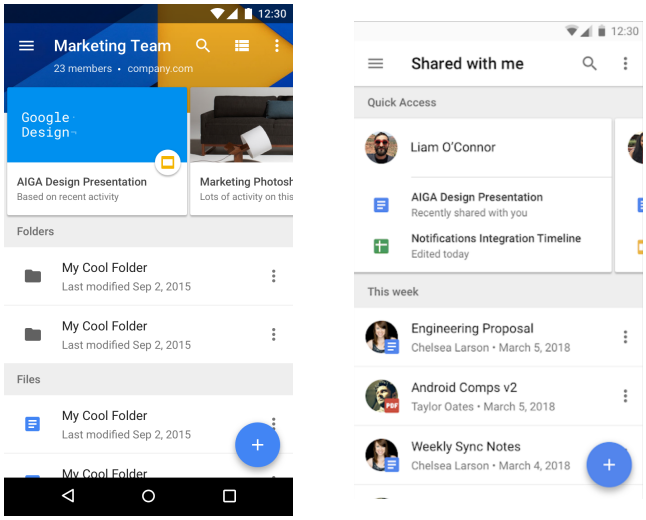

Figure 4: Left: Screenshot of Quick Access suggestions in a Team Drive. Because of the ItemSuggest architecture, suggestions in a Team Drive can share stages and plugins with Quick Access at a user's root, or My Drive, folder. Right: Screenshot of suggestions in the Shared with me view. These suggestions are grouped by collaborator, and ItemSuggest computes the collaborators as a separate application.

Figure 4 shows suggestions on Google Drive's Shared with me view on the right. While those suggestions are also branded "Quick Access," the widget has a different UI treatment, and ItemSuggest evaluates them as a different application triggered via different ListSuggestions call parameters and different structure to the RecordFeedback call. Most importantly, the suggestions are grouped by the people that the user collaborates with. ItemSuggest treats finding the right set of collaborators as a subproblem, and solving that subproblem requires a different set of plugins for candidate creation, feature selection, and ranking. The generic design of ItemSuggest allows the same concepts to apply to suggesting collaborators and suggesting documents. While this may seem obvious in retrospect, this does not reflect the current state of the art in how a majority of prediction infrastructure is built.

Because plugins in ItemSuggest are reusable, the system can also suggest collaborators in other contexts. Figure 5 shows collaborators suggested in a new search box with zero-prefix suggestions. This is useful for enterprise Drive users, who frequently search by person. Being able to reuse entire sets of suggestions allows engineers in ItemSuggest to build features end-to-end faster, which leads to quicker feedback and iterative improvement.

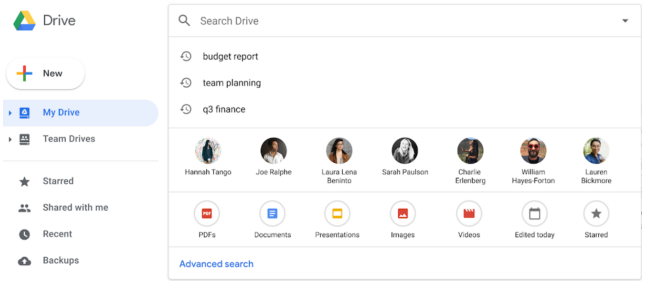

Figure 5: Screenshot of an intelligent search box in Google Drive. In addition to recent searches, we display an intelligent ranking of filetypes and collaborators.

5. RELATED WORK

There is a growing body of work focusing on practical data management challenges in large scale machine learning pipelines. TFX [6] describes many components for analyzing training data, detecting anomalies, and managing the lifecycle of a model drawing on several areas of prior research. While TFX describes a fairly broad set of interoperable tools, in ItemSuggest, we advocate for a much more specific architecture for contextual ranking that simplifies the end-to-end design of the application. ItemSuggest leverages some of the components of TFX such as the model serving stack.

ItemSuggest is closer in spirit to the LASER [4] and Clipper [9] systems. The LASER system deliberately restricts itself to generalized linear models with logistic regression, and prescribes an end-to-end approach to building a platform for such models. ItemSuggest also prescribes a platform along with a common server architecture, but instead of restricting the model class to generalized linear models, we simply rely on the standard interfaces in TensorFlow along with any model that can be efficiently served using TensorFlow serving [19] which incorporates many similar optimizations. ItemSuggest also advocates for a dark-launch and iterate approach where the applications gather their own training data and focuses on providing APIs for efficient feature construction along with model inference. The Clipper system introduces several ideas for managing the latency of inference and allowing applications to react gracefully to degrading models.

Recent literature provides a broad overview of some of the data management challenges in machine learning. They include a systems-focused overview [16] of building ML applications in a database as well as challenges in managing the data in a production system [20]. In contrast to systems like the Data Civilizer [11] ItemSuggest does *not* focus on the data discovery and integration. Discovering and suggesting new sources (and features) that might be relevant to applications on ItemSuggest would be extremely valuable.

Systems like DataXFormer [2] and Wrangler [15] tackle the problem of transformation discovery and specification respectively. They are, however, in the context of data integration. The transformation engines in ItemSuggest are in the context of a feature engineering system for ML and present a very different set of design constraints.

There is also Decision Service [3], a platform that utilizes contextual bandits and abstractions of policy exploration, logging, learning, and deployment in order to allow for online learning of a recommendation system. ItemSuggest is constructed to only allow for supervised learning, though there are future plans to explore the usage of contextual bandits for ranking.

Modeling in ItemSuggest draws on a rich vein of work on learning retrieval functions based on clickthrough data [14] including the use of neural networks [5], and techniques to overcome challenges with traditional approaches including sparsity of clicks and position bias [23]. In addition, growing interest in learning-to-rank from user feedback (e.g. clicks) has resulted in a multitude of literature. [14] presents a canonical approach of using Support Vector Machines to learn retrieval functions based on clickthrough data. [5] incorporates a neural-network based approach to learn from millions of user interactions.

6. SUMMARY AND FUTURE RESEARCH

This paper presented the design for ItemSuggest—a system that allows a data scientist to focus on the core idea–implement–train–evaluate loop and simplifies some of the data wrangling (data acquisition, augmentation, cleaning, transformation to training formats, feature lifecycle management, and experiment infrastructure) that consistutes 80%–90% of an applied ML project. ItemSuggest has many limitations—our design only considers ranking-style problems. While this is an important subset of applied ML projects, many projects don't fit into this abstraction. Also, we don't eliminate the need for all the data engineering, merely reduce it substantially. For instance, adding a new data source still requires a signficant amount of engineering like instrumenting an RPC call to the new source and transforming the response to suitable part of the Candidate protocol buffer. We do hope that the case study of ItemSuggest inspires future work in platforms for building and managing ML applications.

One of the key contributions of this work is in identifying two components of the system which essentially implement a data transformation engine. The feature computation module needs to execute transformations from a set of Candidate protocol buffers and a Context protocol buffer to a set of tf.Example protocol buffers within a given latency budget. The training infrastructure requires a similar transformation component, but the objective here is to support experiment velocity by allowing the data scientist to represent different experiments as different transformations. We believe a well-designed transformation engine could be used for both components and is likely to be useful even beyond platforms like ItemSuggest.

7. ACKNOWLEDGMENTS

Special thanks to Jesse Sterr for his contributions to ItemSuggest. We would also like to acknowledge several engineers for their design and implementation work: Arthur Johnston, Brandon Rodriguez, Brandon Vargo, Brian Calaci, Brian Reinhart, Chih-hao Shen, Chris Walsh, Cleopatra Von-Ludwig, David Gardner, Devaki Hanumante, Divanshu Garg, Hannah Keiler, Jai Gupta, Michael Rose, Ryan Evans, Sean Abraham, Siamak Sobhany, Timothy Vis, Weize Kong, Zac Wilson. The paper also benefited from fruitful discussions with Don Metzler and Marc Najork.

8. REFERENCES

[1] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. Tensorflow: A system for large-scale machine learning. In *OSDI*, pages 265–283, Berkeley, CA, USA, 2016.

[2] Z. Abedjan, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, and M. Stonebraker. Dataxformer: A robust transformation discovery system. In *ICDE*, pages 1134–1145, 2016.

[3] A. Agarwal, S. Bird, M. Cozowicz, L. Hoang, J. Langford, S. Lee, J. Li, D. Melamed, G. Oshri, O. Ribas, S. Sen, and A. Slivkins. A multiworld testing decision service. *CoRR*, abs/1606.03966, 2016.

[4] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. Laser: A scalable response prediction platform for online advertising. In *WSDM*, pages 173–182, 2014.

[5] E. Agichtein, E. Brill, S. Dumais, E. Brill, and S. Dumais. Improving web search ranking by incorporating user behavior. In *SIGIR*, August 2006.

[6] D. Baylor, E. Breck, H.-T. Cheng, N. Fiedel, C. Y. Foo, Z. Haque, S. Haykal, M. Ispir, V. Jain, L. Koc, C. Y. Koo, L. Lew, C. Mewald, A. N. Modi, N. Polyzotis, S. Ramesh, S. Roy, S. E. Whang, M. Wicke, J. Wilkiewicz, X. Zhang, and M. Zinkevich. Tfx: A tensorflow-based production-scale machine learning platform. In *KDD*, 2017.

[7] C. J. Burges. From ranknet to lambdarank to lambdamart: An overview. *Learning*, 11(23-581):81, 2010.

[8] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. Flumejava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[9] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. In *NSDI*, pages 613–627, 2017.

[10] J. Dean. Building machine learning systems that understand. In *SIGMOD*, 2016.

[11] D. Deng, R. C. Fernandez, Z. Abedjan, S. Wang, M. Stonebraker, A. K. Elmagarmid, I. F. Ilyas, S. Madden, M. Ouzzani, and N. Tang. The data civilizer system. In *CIDR*, 2017.

[12] U. M. Fayyad, A. Candel, E. Ariño de la Rubia, S. Pafka, A. Chong, and J.-Y. Lee. Benchmarks and process management in data science: Will we ever get over the mess? In *KDD*, pages 31–32, 2017.

[13] J. M. Hellerstein. People, computers, and the hot mess of real data. In *KDD*, pages 7–7. ACM, 2016.

[14] T. Joachims. Optimizing search engines using clickthrough data. In *KDD*, pages 133–142, New York, NY, USA, 2002. ACM.

[15] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer. Wrangler: Interactive visual specification of data transformation scripts. In *SIGCHI*, pages 3363–3372, 2011.

[16] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD*, pages 1717–1722. ACM, 2017.

[17] S. Lohr. For big-data scientists,'janitor work' is key hurdle to insights. *New York Times*, 17, 2014.

[18] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ML serving. *CoRR*, abs/1712.06139, 2017.

[19] C. Olston, N. Fiedel, K. Gorovoy, J. Harmsen, L. Lao, F. Li, V. Rajashekhar, S. Ramesh, and J. Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[20] N. Polyzotis, S. Roy, S. E. Whang, and M. Zinkevich. Data management challenges in production machine learning. In *SIGMOD*, pages 1723–1726. ACM, 2017.

[21] S. Tata, A. Popescul, M. Najork, M. Colagrosso, J. Gibbons, A. Green, A. Mah, M. Smith, D. Garg, C. Meyer, and R. Kan. Quick Access: Building a smart experience for Google Drive. In *KDD*.

[22] K. Varda. Protocol buffers: Google's data interchange format. *Google Open Source Blog*, 72, 2008.

[23] X. Wang, M. Bendersky, D. Metzler, and M. Najork. Learning to rank with selection bias in personal search. In *SIGIR*, pages 115–124, 2016.