Improving Recommendation Quality in Google Drive

Suming J. Chen, Zhen Qin, Zac Wilson, Brian Calaci, Michael Rose, Ryan Evans, Sean Abraham, Donald Metzler, Sandeep Tata, Mike Colagrosso
Google LLC, USA
{suming,zhenqin,zacwilson,calaci,mrrose,ryanevans,seanabraham,metzler,tata,mcolagrosso}@google.com

ABSTRACT

Quick Access is a machine-learned system in Google Drive that predicts which files a user wants to open. Adding Quick Access recommendations to the Drive homepage cut the amount of time that users spend locating their files in half. Aggregated over the ~1 billion users of Drive, the time saved up adds up to ~1000 work weeks every day. In this paper, we discuss both the challenges of iteratively improving the quality of a personal recommendation system as well as the variety of approaches that we took in order to improve this feature. We explored different deep network architectures, novel modeling techniques, additional data sources, and the effects of latency and biases in the UX. We share both pitfalls as well as successes in our attempts to improve this product, and also discuss how we scaled and managed the complexity of the system. We believe that these insights will be especially useful to those who are working with private corpora as well as those who are building a large-scale production recommendation system.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; • **Computing methodologies** → **Neural networks**.

KEYWORDS

recommender system; deep learning

ACM Reference Format:
Suming J. Chen, Zhen Qin, Zac Wilson, Brian Calaci, Michael Rose, Ryan Evans, Sean Abraham, Donald Metzler, Sandeep Tata, Mike Colagrosso. 2020. Improving Recommendation Quality in Google Drive. In *Proceedings of the 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '20), August 23–27, 2020, Virtual Event, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394486.3403341

1 INTRODUCTION

Quick Access [28] is a feature in both the mobile and web versions of Google Drive that provides users a shortcut to their most relevant files. Improving Quick Access recommendations is important because of the massive amount of traffic received—O(millions) of clicks a day. As [28] has demonstrated, using Quick Access gets users to their files around 50% faster than if they had not. Those time savings add up: Quick Access currently saves users 1000 work weeks every day. That's a 20-year career worth of work, every day.

Quick Access differs from a traditional recommendation system in that the items for recommendation are files selected from a user's private corpus. Since this corpus is queried in real-time given the user's context (e.g. the last 100 files the user viewed), we can look at Quick Access as a special case of a context-aware recommendation system [3] where even the candidates and their features for ranking are generated from the context. That being said, we found that even for this case of building a *personal recommendation system*, that candidate generation, modeling, and ranking techniques used for various other recommendation systems [10, 22] are still applicable.

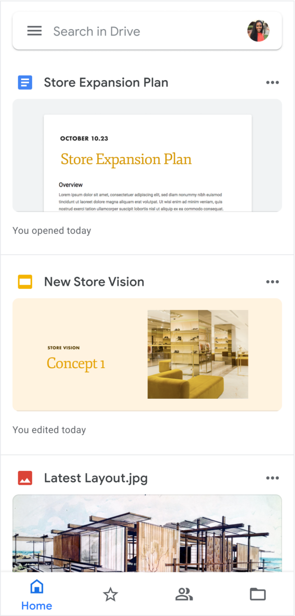


Figure 1: The redesigned Drive app features Quick Access as the Home view.

In this paper, we describe a variety of approaches used to improve the quality of recommendations for Quick Access and show that click-through rate (CTR) on Quick Access is 34.8% better, and the accuracy (defined as whether or not a user clicked a file the model predicted, regardless of if the user clicked the Quick Access feature) of predictions is 10.7% better.

From our different approaches, we saw that infrastructure work to improve serving latency had the most impact.[1] We saw that traditional feature engineering did help, but we found that there was also significant improvement stemming from changing the

KDD '20, August 23–27, 2020, Virtual Event, CA, USA

ACM ISBN 978-1-4503-7998-4/20/08.
https://doi.org/10.1145/3394486.3403341

[1] Improving serving latency affects CTR only, which explains why overall Accuracy didn't increase as much as CTR.

deep neural network (DNN) architecture for the ranking model. We believe that the lessons learned in this work are widely applicable for those working on recommendation systems, especially on 1) private corpora where techniques like matrix factorization aren't available (due to the sensitivity of the data) and on 2) large scale production systems where the infrastructure needs to support millions of requests each day.

We first give a brief initial system overview of Quick Access in Section 2, describing the state before we started iterating on quality. Next, we give a comprehensive overview of different techniques that we took to improve the quality of Quick Access in Section 3, covering candidate generation, deep network optimizations, serving infrastructure optimization, various modeling approaches, and feature engineering. We then present metrics demonstrating our progress in quality development in Section 4. Following that, we discuss scaling, reducing model complexity, and improving quality iteration processes in Section 5. We then discuss related work and conclude the paper with some planned future work.

2 INITIAL SYSTEM OVERVIEW

At a high-level, Quick Access [28] works as follows and the steps can be seen in Figure 2:

(1) When a user visits Google Drive, requests are made to various data stores to collect contextual information such as the user's activity in Drive over the past few weeks, what files are attached to recently sent Gmail messages, and what files are attached to Calendar meetings. This is the start of the user's *session.*

(2) The contextual information from the first stage is transformed and processed to create contextual features, candidates, and per-candidate features to be ingested by a machine-learned (ML) ranker. These candidates are selected to consist of the most recent files, as [28] showed that over 75% of *open* events occur on files with some activity in the past 60 days.

(3) The ML model ranks candidates using the generated features. Then results are fine-tuned with business logic. For instance, some business logic generates an "explanation" for why a file is suggested. This explanation is visually displayed to the user.[2]

(4) The results are served to a user and the session ends when a user clicks on a document anywhere within Drive. High-level click metadata is sent to the ML pipeline.

(5) ML pipeline incorporates the candidates, corresponding features, and click information to train a new model which can replace the existing production model in a one-off manner.

Quick Access utilizes a deep network implementation for the ranking model, similar to other production recommendation systems [10, 14]. The initial model was trained using TensorFlow [1] and utilized a pointwise learning-to-rank approach [23]— where for every user session, given the n candidates that were generated to be ranked, the first user-clicked file is given a positive label and the other $n - 1$ unclicked files given negative labels, *no matter* if the clicked file was clicked via the Quick Access UI or some other location within Drive. Thus, the ranker is modeled as a pointwise binary classification problem.

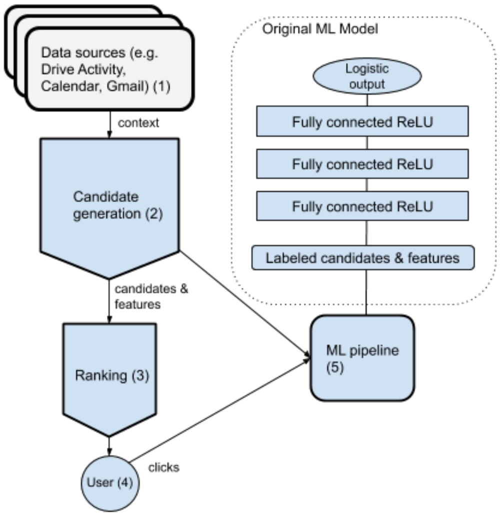


Figure 2: Simple recommendation system overview. Key problems are A.) how to take the generic data and select candidates, and B.) how to rank those candidates. Based on implicit user click feedback on the generated candidates, new machine-learned (ML) models can be trained.

The input feature vector to the model contains a variety of features constructed from streams of user activity on Drive—these streams are represented as sparse, fixed-width vectors where events through the day are bucketed into the vector [28]. The original model had the following high-level categories of features:

- Histograms of user event counts.
- Histograms of user usage across various categories (e.g. different file mime types).
- Time series of user events across client types (e.g. web vs. iOS).
- One-hot encodings of various rank features, e.g. how the file ranks in terms of recency.

3 QUALITY IMPROVEMENT WORK

We took several approaches to improve quality:

- improving the candidate generation process
- improving the architecture of the deep model beyond a simple stack of fully connected layers and ReLUs
- improving our model representation, e.g. accounting for position bias and cleaning/filtering training data
- feature engineering
- improving serving latency to improve engagement

Note that for any changes dealing with training a new ML model, we have guidelines that state that newly trained models can launch only given the following criteria:

- Improvement in core metrics like *accuracy* and *CTR.*[3] Unless otherwise specified, *all* presented experimental results are *statistically significant.*

[2] For example, in Figure 1, see the explanation string that reads "You opened today".

[3] Even relatively small improvements are approved, as long as they are statistically significant. The additive compound effect is evident over time.

- The experimental model has been in a live A/B experiment for at least 1 week.[4]
- Minimal increase in latency.
- Minimal increase in training complexity. Training a model shouldn't take much longer or require a series of steps that is difficult for other engineers to replicate.

3.1 Candidate Generation

Originally, files with user Drive activity in the past 60 days were considered as candidates for ranking. In order to reduce the amount of processing and storage that ranking these candidates would require, we set a limit of 500 candidates using only the past 40 days of activity data. This change resulted in no losses in our metrics. In order to further reduce the number of processed candidates and to deal with users who had sparse activity for the past 40 days, we experimented with moving to a cap of 100 candidates using the past 60 days of activity data. We believe that there was little coverage from an additional candidate pool due to the importance of recency.

Our work resulted in the following decreases in latency with no adverse effects on quality metrics:

- 3% decrease at the 50th percentile,
- 15% decrease at the 90th percentile
- 24% decrease at the 99th percentile.

Candidate generation experiments led to *recall* being tracked and evaluated as a core metric for any launch decision. Recall for a candidate generation scheme is defined as:

$$Recall = \frac{\text{\# sessions s.t. initial click on } C \in C'}{\text{\# of sessions with a click}} \tag{1}$$

where C' is the set of candidates generated for ranking in that session and clicks *anywhere* within Drive are counted.

Since the recall numbers generally started fairly high, we believed there was limited headroom of using an ML model. We instead decided to use heuristics to incorporate additional sources of candidates. This has led to different degrees of improvement. For instance, adding in files that were shared to the user or where a user was mentioned by comment led to as much as a 5.3% relative jump for certain user groups. Conversely, adding in files that were found as attachments in a user's Gmail account only led to a 0.2% relative jump.

3.2 Deep Network Optimizations

We share how we adapted some recent deep learning techniques, ranging from changing loss functions to making network architecture changes, each of which led to significant online metrics gains. Note that a diagram of the initial Quick Access network architecture can be seen in Figure 2.

Changing Loss Functions. The initial launched model was trained using log-loss as the loss function and AUCLoss as the evaluation function. However, due to concerns about data imbalance where each session could potentially contain up to 99 negative examples and only 1 positive example, we moved to using AUC Precision-Recall (AUC-PR) as both the offline evaluation function and loss function [13]. This resulted in a 0.45% increase in our offline AUC-PR numbers, which led to a relative increase of 1.0% and 0.6% in live Accuracy and CTR metrics.

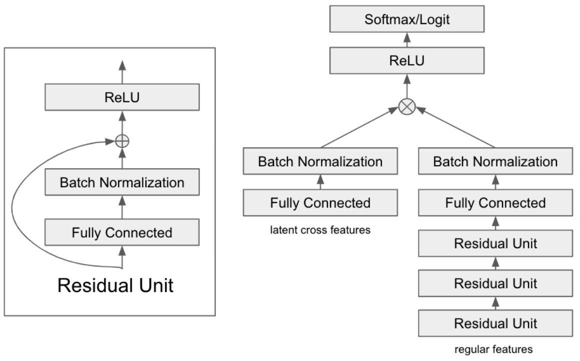


Figure 3: Second-stage model architecture (after latent cross, residual units, and batch normalization).

Latent cross. Latent Cross [5] is a recent technique deployed in YouTube's Recurrent Neural Network (RNN)-based recommendation systems. It feeds certain features into a separate shortcut and allows for more efficient higher order feature interaction learning. For example, in recommendation systems, it makes sense to implicitly generate feature interactions between user type and other candidate features. We took this approach as we saw that other popular recommendation frameworks [21] also use feature cross approaches similar in spirit to great success.

$$f_{\text{cross}} = (1 + w_{\text{latent}}) \odot h_{\text{out}},$$

where w_{latent} is the output of the latent cross path, $\odot$ is a simple element-wise multiplication, and h_{out} is the output of the final regular deep layer. We adapted it to the feedforward network architecture as shown in Figure 3. The features we include as latent cross features are user type (User Group 1 vs. User Group 2)[5], platform (mobile vs. desktop), and day of the week. All these features possess low dimensionality as suggested in the original paper.

Though we found that there was minimal gain in offline metrics and evaluation, in live experiments we found that there was significant gain for all experiment groups. Given that the largest gains came from User Group 2 mobile users, our belief is that utilizing latent cross enabled us to improve metrics for small groups (e.g. User Group 2 mobile users) in training data.[6]

ResNet. Residual units [17] are very popular building blocks for computer vision problems with convolutional filters. They contain shortcut links that facilitate gradient flow and thus optimization.

We adapted residual units to our feedforward network and stacked multiple units together as shown in Figure 3. Note that our residual

[4] Weekend metrics are different than weekday metrics, so we eliminate this bias.

[5] Group information omitted to protect proprietary information.

[6] Due to various policy restrictions, which is further discussed in Section 5, we face a significant lack of data from User Group 2 users.

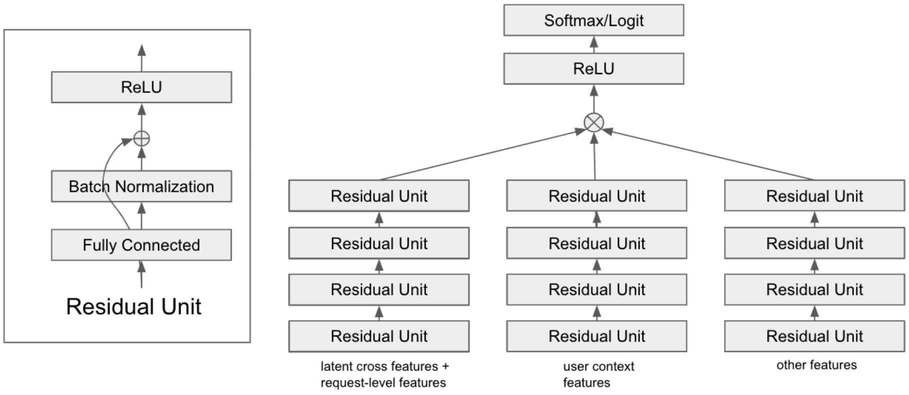


Figure 4: Final model architecture.

unit only contains one fully connected layer, while in the computer vision community it is common to use two layers. We did not observe clear gains by trying two-layer units. Our hypothesis is that while two-layer units are crucial for computer vision problems, this does not translate to large-scale recommender system datasets like ours. In experiments we show that ResNet significantly improves metrics with minimal extra complexity, leading to relative increases of 0.56% Accuracy and 0.45% CTR for web User Group 2 users.

Batch normalization. Batch normalization (BN) is a generic technique in deep learning that accelerates convergence and provides regularization [18]. We adapted BN as shown in Figure 3. We also observed multiple benefits including faster convergence and better testing performance.

One thing we did not include is input layer BN (i.e., BN before the first fully connected layers.) We note that it significantly increases training time - each training step takes about 3 times longer due to the large number of input features we deal with. We were able to train a model with input layer BN and observe that, though validation loss decreased faster per step during early steps, each step took longer and the performance plateaued quickly and ended up without clear gains. Thus, we recommend that practitioners building large scale recommender systems with many features also consider dropping input layer BN.

Deep & Cross Networks. As the previously introduced latent cross feature interactions proved beneficial, we further improved upon the architecture shown in Figure 3 by experimenting with models based on the Deep & Cross Network (DCN) architecture [29]. DCN, similarly to latent cross, allows for more efficiently learning feature interactions by grouping semantically similar features into towers and crossing them at the end.

We investigated how to partition the features – we emphasized having clear guidelines semantically of how to partition, to ensure that future contributors who added new features would be able to intuitively know which tower to add their features to. Therefore, we experimented with both two and three tower models, and found that the most significant offline gain came from splitting the features into three towers, with 1) one tower including all previous latent cross features as well as additional request-level features, 2) one tower including file-level user contextual features, such as if a candidate contains certain contacts and n-grams that are relevant to the user, and 3) all other features. The downside of this approach is that it nearly doubled our model training time.. The final model architecture can be seen in Figure 4, leading to relative increases of 1.0% Accuracy for web User Group 2 users and a 1.03% CTR for Android User Group 2 users, both significant increases.

3.3 Modeling Approaches

Candidate sampling. In Section 5 we discuss how we moved from a pointwise loss model to a listwise loss model in depth, but one of the decisions we had to make was how many of the 100 candidates to sample as the negative examples when computing listwise loss per example. Table 1 shows some accuracy losses for different sampling schemes. Initially we thought that the majority of the negative examples would not provide additional signal, so only a few would need to be sampled in order to better facilitate learning. This proved to be completely incorrect and in the end we found that including *every* candidate in the training process was critical to maintain accuracy numbers.

Position bias. Given that around 95% of clicks on the Quick Access feature are on the top 3 files shown, we wanted to address the *position bias* that might be impacting our training process. We used an approach proposed by [32], where the file position is fed in during training with feature drop-out to prevent overreliance. We

# of candidates sampled	Relative accuracy loss
10	30.9%
25	4.2%
50	1.3%
100	0.0%

Table 1: Relative offline accuracy loss of sampling different number of negative examples.

found that this technique did not lead to any significant metrics increase and believe that this is due to half of our training data coming from clicks outside of Quick Access, thus rendering the method of [32] less effective. We did see some success in applying EM-based techniques, and further discuss this in [25].

Training data integrity. There were several motivations behind wanting to clean and filter our training data in order to increase its integrity. Recall that each example in our training data is generated from a user's session in Drive, where a user visits Drive and opens a file. However, the features may be populated from any interactions with the files, inside or outside of Drive.[7] The belief we hold is that by modeling the clicked files as positive examples and unclicked files as negative examples, we can learn how to predict relevant files at serving time. This led us to investigate if we could improve upon this modeling by cleaning out potential sources of noise, as we had the following concerns:

(1) What if a user clicks on a file a significant time after the predictions were shown?
(2) What if a user clicks on a file outside of the Quick Access UI?
(3) What if a user's files do not have interactions that capture the true relevance of the file to the user?

To address (1), we hypothesized that there is some direct correspondence between the integrity of an example and how much time had elapsed since the click. We found that over 10% of the examples we were computing had a click that occurred more than 1 minute after the user loaded the Drive page, and of these examples, roughly 50% occurred more than 5 minutes after. As the training data contains features computed at the *beginning* of the session, our belief was that filtering out these "delayed-click" examples would lead to learning a more representative ranking model. We tried both filtering out examples with over a 1 minute delay and 5 minute delay, but found no difference in model quality, even accounting for the reduced training data set size.

For (2), since we collect clicks on files from both the Quick Access UI and elsewhere in Drive, we also measured the effect of *only* training with clicks from the Quick Access UI, due to the concern about potential noise from clicks outside the UI. We generated two training data sets, a dataset $D_{filtered}$ with filtered clicks from only the Quick Access UI, and a sampled dataset $D_{sampled}$ with clicks from all sources, ensuring that $|D_{filtered}| = |D_{sampled}|$. We saw that using only the filtered set led to a 0.6% relative decrease in Accuracy. From the results of this experiment, we can confirm that this *exploratory* data is useful and helps reduce the problem of having a confounding training "feedback loop"as described in [7]. We believe this warrants further investigation in how to provide less biased data for training a model.

For (3), we took a high-level look at *what kind of events* were being counted as interactions. We found that we could categorize the events in Drive activity into two buckets:

- user events – events that were directly triggered by a user, e.g. reading or editing a file.
- "third-party" events – events that were triggered by some third party, e.g. a sync client.

We suspected that events of the latter type could be causing significant noise as 1) they don't correspond to user action and 2) they are often done in bulk. By coming up with some heuristic rules to classify events into one of the above categories, we found that 24% of events could be considered "third-party", and we filtered them out when building features. Using the new feature streams resulted in a 1.3% increase in relative CTR, and this new filtered stream of events became the default to compute features from.

Training group-specific models. As mentioned in Section 3.2, we have multiple groups of users that we care about. We filtered out all non-User Group 2 data and trained a model only specifically targeting User Group 2 in the hope that this targeted model would result in better numbers. We found that unsurprisingly, this model performed significantly worse for User Group 1, but somewhat surprisingly, that this trained model performed no better for User Group 2 than the production model. Our belief is that given that the production model already knows if the user is in User Group 1 or User Group 2 (e.g. facilitated by the latent cross we added), so the production model can thus tailor its prediction to that category of user. Training on group specific data provided no additional signal relative to training on all data from all users with a feature indicating the user's group.

3.4 Feature Additions

The majority of the quality effort came from adding new input data sources and engineering new features to add to the model. We attempted to craft features that would provide us a new source of signal that was not redundant with the existing model. Producing experimental models involved 1) adding new data sources, 2) engineering and logging new features, 3) training and tuning models with the added features. Through all our iterations, we estimate that only about 15% of the feature sets that we have experimented in have resulted in a significant improvement and model launch.

In this section, we give an overview of the various groups of features whose integration successfully increased the quality metrics for the model. We show metrics for different user groups of interest, ranging from User Group 1/User Group 2 to web/mobile users in Table 2. After discussing some of the successful feature launches, we also discuss some efforts that didn't end up working and share our intuition on why.

Collaborator & upload features. We added a file-level feature on whether a document was also shared with a user's recent collaborators as well as some other features related to the recent file upload timestamps for the user, and in Table 2 we can see that this was particularly beneficial for User Group 2 users, whom we know have more collaborators in general than User Group 1 users.

File metadata features. Whereas the sole previous source of features was the user's stream of activity in Drive, we integrated a new data source that would allow us to get some additional metadata about a Drive file, including information about whether or not the file was starred, when it was last modified by the user, if the user has edit permission, etc.

[7] For example, a user may have a sync client installed that automatically uploads recently-added files to Drive.

Feature Set	Metrics	User Group 2 Web	User Group 2 Mobile	User Group 1 Web	User Group 1 Mobile
(1)	Accuracy	1.70%	0.38%	1.09%	0.59%
	CTR	0.52%	0.63%	0.62%	0.58%
(2)	Accuracy	2.48%	-0.05%	2.10%	0.57%
	CTR	2.08%	0.00%	1.83%	0.67%
(3)	Accuracy	0.94%	1.78%	0.78%	0.76%
	CTR	0.45%	0.89%	0.64%	0.45%
(4)	Accuracy	0.27%	0.59%	0.09%	0.15%
	CTR	-0.27%	1.19%	0.03%	0.20%
(5)	Accuracy	0.66%	0.84%	0.37%	0.36%
	CTR	0.55%	0.25%	0.66%	0.49%

Table 2: Metrics improvements from various feature additions. All numbers reported are relative improvements. Statistically non-significant results have been greyed out.

From this data source, we crafted a variety of time-based and yes/no features representing the file metadata described above. For this case, even though we found that retrieving this new data source increased our latency, the quality gain was so substantial that we made an exception and still rolled out these features.

Platform feature. We added a simple feature related to what platform the request to Drive came from, e.g. web, iOS, etc. Interestingly, we found that just adding this feature wasn't very effective in our offline evaluation, which prompted exploring the uses of other architectures. See Section 3.2 for more details. As expected, we saw a very significant jump in metrics for mobile users.

External Context features. We added a collection of signals capturing user-specific contextual data such as which contacts, files, and n-grams are important to a user.

User features. This collection of features consists of:

- A user type feature which segments users into either User Group 1 or User Group 2 categorization.
- User time features which define the hour of day and day of week in the user's local time zone.
- Calendar features indicating time to user's next meetings and attached files.

Pitfalls: Features that didn't work. There were multiple instances where we engineered sets of features that we were confident would bring gain but instead led to neutral metrics. After extensive debugging for the features that didn't work out, we offer the two explanations as to *why*:

First, given that each candidate example initially contains tens of thousands of floats and we had little intuition about the semantic meaning of existing features, we often found that the features that we were attempting to add already had some redundancy. For example, we experimented with adding features such as "number of Drive sessions a day" because we wanted to capture the behavior of power users. However, upon further examination of our existing features we found that this feature was only derivative. This has since led us to prioritizing documenting existing features and reducing model complexity, as detailed in Section 5.

Second, we have problems of 1) collecting biased feedback via our existing ranker and 2) sparsity in signal for various candidates—certain candidates are missing features. As stated in Section 2, candidates and their features are created from a variety of sources, the primary source of features for ranking coming from activity in Drive. Candidates from other sources (e.g. Gmail, Calendar) may not have any Drive activity and would thus not be properly ranked by the ML model. We saw an example of this manifest when we added Gmail-related signals to the model—our guess is that even though they would benefit candidates generated from Gmail data, since those candidates can never be ranked in the top k, we can't collect sufficient feedback in order to compensate for those candidates missing existing ranker features.

3.5 Serving latency optimization

In addition to model improvements, we worked to improve the serving latency of the Quick Access system.[8] We spent a lot of time optimizing server request latency, but ultimately the majority of our user-perceived latency for our web application originated from front-end rendering. The Drive front-end is a large JavaScript-based web application, which must be loaded, parsed, and executed from the network (or cache), and then initialized to run handlers to begin requesting data. Because this process must complete before any request for Quick Access recommendations could be made, users would perceive significant delay for Quick Access.

To improve the latency, we made an infrastructure change to send a server-side request to begin computing Quick Access predictions as soon as a user loads the Drive homepage. As the user's browser parses and renders the website, we're able to complete our request in parallel and stream the prediction results to the user. The server renders the initial page, flushes that to the user, and keeps the initial page socket open, avoiding sending the HTML body closing tag. Once predictions have been loaded, a <script> tag is injected into the page with the results with a callback to render the predictions and closing the HTML document tags.

Overall, we observed large improvements in end to end render times (time from page open to Quick Access render):

[8] Though model evaluation itself is extremely fast in Quick Access, collecting candidate documents, computing features, and checking user access permissions for those documents takes considerably more time.

- 18.5% improvement at the 50th percentile.
- 6.5% improvement at the 95th percentile.
- 23.1% improvement at the 99th percentile.

Overall, we observed large improvements in end to end render times (time from page open to Quick Access render), with 18.5% and 23.1% improvement respectively at the 50th and 99th percentile. This translated into very significant relative increases of **5.06%** in CTR for User Group 2 users and **5.38%** in CTR for User Group 1 users. In addition, the time it takes a user to open a document decreased 12.3% for User Group 2 and 11.7% for User Group 1.

4 METRICS

Our quality improvement work has led to significant product gains. Figures 5a, 5b, 5c respectively show the CTR, Accuracy, and Recall changes between the ML model and the MRU heuristic over a period of ~18 months. Note that we omit the absolute numbers on the figures' y-axes to protect proprietary information. The figures compare our production machine learning model, labeled ML, to a baseline recommender, labeled MRU. MRU stands for *most recently used*, and serves the user's most recent files, regardless of context. It's important to have such a baseline for comparison, so we maintain a persistent experiment to serve MRU recommendations.

We can see from these figures that though there is some noise in the metrics[9] they have steadily increased over time. Moreover, the ML model's advantage over the baseline MRU recommendations has consistently increased. The steady increase has come from system improvements we have made to optimize for *experiment velocity*. Just as we compare our ML model to MRU in Figures 5a, 5b, 5c, our system automatically labels, evaluates, and plots every experimental model in comparison to the production model. We have found that investing upfront in removing the overhead needed to build and launch a new experiment leads not only to more experiments, but to more ambitious experiments from a wider set of collaborators. Those improvements from those experiments have stacked over time for a relative CTR and accuracy increases of **34.8%** and **10.7%**.

Even Recall, which as we discussed was already fairly high, has improved via our candidate generation approaches. Both (1) not limiting candidates to files recently accessed by the users and (2) including files from Gmail, Calendar, and shared files updated by the user's collaborators has led to additional useful candidates.

5 SCALING AND COMPLEXITY REDUCTION

In this section we discuss some various techniques we took in order to scale the system and training infrastructure.

Infrastructure changes. As discussed in [27], Quick Access was moved onto ItemSuggest—a framework that collects training data at prediction time, eliminating train-serve skew. However, a downside of this approach is that it slows down the training process because features have to be implemented in ItemSuggest before they can be used. We found that implementing offline pipelines to do feature transformations acted as a fair compromise that allowed us to avoid train-serve skew while 1) still allowing more flexibility in training and 2) preventing the accumulation of too much technical debt in the critical serving path.

For the ML model itself, we moved from a vanilla TensorFlow [1] model to TF-Ranking [24], which allowed us to train models with listwise loss [6], reducing training data size by approximately 50%, giving us access to offline ranking metrics, and reducing the time to train a model by over 50%.

Whereas the initial launched model had hyperparameters selected by running grid search, for subsequent model improvements we considered methods similar to those found in [8] before settling on Vizier [15]. Vizier is a black-box optimization service which reduces the number of necessary trials to tune hyperparameters such as the number of layers, width, initialization weights, and even which loss functions are utilized.

In addition, we moved the model training pipeline to TFX [4], which allowed for automatic continuous retraining to guarantee that we always have an up-to-date model trained on fresh data. There were several other benefits we observed, such as 1) automatic anomaly detection and skew detection, 2) model verification to sanity check and automatically ensure that a newly trained model meets some quality threshold, and 3) increased ability to evaluate models over subsets of data.

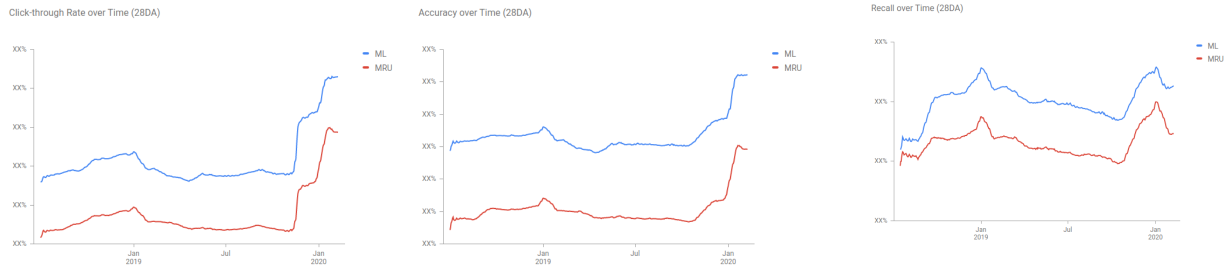
Feature reconfiguration. The initial Quick Access model had 38k floats per candidate, including individual features that consisted of large numbers of floats. This causes difficulties and slowed down quality development because 1) feature ablation studies are more difficult, 2) it is more difficult to catch bugs in the features, 3) it is difficult to apply certain DNN tricks (e.g. latent cross). Also, without strong semantic knowledge of existing features, when adding new features fails to improve metrics, it is unclear whether 1) the new features have bugs, 2) there is redundancy with existing features, 3) the new features actually are not useful. Our previous approach did track the mean and variance for certain feature values, but as the initial model had feature vectors of 38K floats per candidate, it quickly became infeasible to use this approach due to the large sparse vectors, and multiple bugs went undetected.

Significant work was done to reconfigure the feature generators to output smaller blocks of features so that we could run experiments and use [15] along with other feature importance approaches to prune the features that the model ingests. Identifying redundant features and reconfiguring the inefficient feature generators allowed us to remove around 34k floats in total, leading to a final model needing around just approximately 4k floats per candidate.

Improving model training workflow. Reproducibility in ML model training has been a challenge we consistently take care to address. We have seen cases where a model trained on fresh data (with the same hyperparameters and features) can outperform the existing production model. We have also seen cases where for several months the existing production model could not be replaced. Part of this is due to the seasonal behavior of users, e.g. how during the summer there is less activity from students.

Compounding the problem of reproducibility is the sensitive nature of the data. Policy constraints restrict the length of time we can use a dataset. The ephemeral nature of our data makes reproducibility difficult—there have been cases where multiple collaborators are taking different quality approaches and we have difficulty deciding

[9] Which could be attributed to weekend vs weekday metrics, seasonal behavior like school being in session, other new features being added to Drive.



(a) CTR

(b) Accuracy

(c) Recall

Figure 5: Improvements of CTR (a), Accuracy (b), and Recall (c) over a period of ~18 months, showing the difference between the ML model and the MRU heuristic. Note that y-axis numbers have been deliberately omitted to protect proprietary information.

between which model should be launched. If we wanted to revisit the data and see if another technique would have produced different results, we cannot because the data has since expired.

We took two steps to help improve reproducibility issues. The first is when training an experimental model, to always retrain a new baseline model that has the same hyperparameters and features as the production model. This helps us determine how much to take seasonality into account and increase experimental rigor. The second is to have various practitioners all trying their experimental approaches on the same generated baseline to decrease both the variance in the observed results and the number of baseline experiments needed. Though these approaches don't completely solve the reproducibility challenge, we have observed that as a result we require running fewer live experiments on different ML models in the quality development process.

Attempting to move away from a deep model. Finally, we made an unsuccessful effort to replace the deep model with a *gradient boosted decision tree* (GBDT) model. There are uses of both GBDT and deep networks in the industry, with [22, 30] opting to use GDBT and [10, 16] picking deep networks. The combination of GBDT and deep networks is also studied [20], but it is not easy to adjust the serving infrastructure to adapt both.

In an effort to bring about more model interpretability, latency wins, better support for feature engineering, and more frequent iterations of model development (due to how much data is typically needed for a deep network), we devoted some time to training GDBT models. However, we found that due to the high number of pre-existing sparse features, even with significant feature engineering effort we were unable to match the quality of the deep network. We also hypothesize that because of the large feature vector sizes, our infrastructure for GDBT training was not capable of learning as many cross feature interactions. Luckily, through this GDBT effort we were able to dig more deeply into the feature generation logic and found redundancies that led to several critical bug fixes.

6 RELATED WORK

For other industry case studies of recommendation systems, [10] discusses work on improving YouTube's recommendation system. [22] covers the gradual improvement of the Pinterest "Related Pins" recommendation system. This work also uses learning-to-rank to boost the quality of the system. [14] discusses deep learning recommenders at the popular Norwegian website FINN.no, where they use multi-armed bandits as a high-level reranker on top of other recommendations. For search, [16] gives an in-depth look at how deep learning enabled them to significantly improve Airbnb search.

For other work on improving network architecture for recommendation systems, [9] introduces framework for jointly training linear model alongside a DNN, which is similar in spirit to the DCN approach [29], except feature crossing is done on original features instead of embedded features.[10][3] discusses how recommendation systems can make use of contextual information for making more intelligent recommendations, and this has been further explored by [5] for application within YouTube. In terms of the bias present in recommendation systems, [7] demonstrates the adverse effects of bias in recommendation systems in leading to homogenous behavior. [2] proposes sanitizing biased feedback and changing product interface to reduce bias. [26] utilizes causal inference techniques with matrix factorization techniques to address the issue.

[31] offers an extensive survey of deep learning based recommender systems and includes a lengthy discussion of the advantages and disadvantages of utilizing deep learning techniques for recommendation systems. There is also significant work on how to improve the architecture of various recommender systems. [11] proposes using RNNs to capture the evolution of users' preferences to the files' latent features. Our modeling may weakly encode changing user preferences, but directly modeling the historical interactions may prove more promising.

7 CONCLUSIONS AND FUTURE WORK

In this work, we reviewed a significant effort in quality iteration for a personal recommendation system. We provided examples of successful modeling techniques as well as pitfalls, and hope that they provide a useful case study for those interested in building recommendation systems over private corpora. We found that of all the various quality work, that infrastructure work to reduce latency had the most impact. Other approaches like feature engineering and changing DNN architecture also led to quality wins. We found that for certain features, DNN architecture needed to be changed in order

[10] As our starting infrastructure consisted solely of sparse features, we did not experiment with this framework.

to fully utilize those features. We confirm that the usage of DNN was able to take advantage of a lot of sparse, difficult-to-feature-engineer features in a way that approaches like decision trees can't. Changes to try to explicitly model "good" user interactions (e.g. filtering out clicks by how much delay there was) didn't help.

We also want to stress the importance of simplifying the model, improving the infrastructure, and improving model training workflow as we found that having a less complex model and better reproducibility in model training could allow for many more data science practitioners to ramp up and contribute.

For future work we plan to investigate optimizing for sequential recommendations, similar to [12], as currently our modeling and evaluation is completely biased towards only caring about the user's first file open. We also plan to investigate the usage of multi-armed bandits [19] in order to deal with our problem of getting feedback for candidates with especially sparse features.

8 ACKNOWLEDGEMENTS

We acknowledge the work of multiple engineers and researchers who were also instrumental in improving this product: Jesse Sterr, Divanshu Garg, Weize Kong, Yan Zhu, Suxin Guo, Rama Kumar Pasumarthi, Nadav Golbandi, and Vlad Panait. We also thank Xuanhui Wang for his feedback regarding this paper.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] Gediminas Adomavicius, Jesse Bockstedt, Shawn Curley, and Jingjing Zhang. 2014. De-biasing user preference ratings in recommender systems. In *RecSys 2014 Workshop on Interfaces and Human Decision Making for Recommender Systems (IntRS 2014)*. 2–9.

[3] Gediminas Adomavicius and Alexander Tuzhilin. 2008. Context-aware Recommender Systems. In *RecSys*. 335–336.

[4] Denis Baylor, Eric Breck, Heng-Tze Cheng, Noah Fiedel, Chuan Yu Foo, Zakaria Haque, Salem Haykal, Mustafa Ispir, Vihan Jain, Levent Koc, Chiu Yuen Koo, Lukasz Lew, Clemens Mewald, Akshay Naresh Modi, Neoklis Polyzotis, Sukriti Ramesh, Sudip Roy, Steven Euijong Whang, Martin Wicke, Jarek Wilkiewicz, Xin Zhang, and Martin Zinkevich. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD*.

[5] Alex Beutel, Paul Covington, Sagar Jain, Can Xu, Jia Li, Vince Gatto, and Ed H. Chi. 2018. Latent Cross: Making Use of Context in Recurrent Recommender Systems. In *WSDM*. 46–54.

[6] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. 2007. Learning to Rank: From Pairwise Approach to Listwise Approach. In *ICML*. 129–136.

[7] Allison J. B. Chaney, Brandon M. Stewart, and Barbara E. Engelhardt. 2018. How Algorithmic Confounding in Recommendation Systems Increases Homogeneity and Decreases Utility. In *RecSys*. 224–232.

[8] Suming J Chen, Xuanhui Wang, Zhen Qin, and Donald Metzler. 2020. Parameter Tuning in Personal Search Systems. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 97–105.

[9] Heng-Tze Cheng, Levent Koc, Jeremiah Harmsen, Tal Shaked, Tushar Chandra, Hrishi Aradhye, Glen Anderson, Greg Corrado, Wei Chai, Mustafa Ispir, et al. 2016. Wide & deep learning for recommender systems. In *Proceedings of the 1st workshop on deep learning for recommender systems*. 7–10.

[10] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep Neural Networks for YouTube Recommendations. In *RecSys*.

[11] Hanjun Dai, Yichen Wang, Rakshit Trivedi, and Le Song. 2016. Recurrent Coevolutionary Latent Feature Processes for Continuous-Time Recommendation. In *Proceedings of the 1st Workshop on Deep Learning for Recommender Systems*. 29–34.

[12] Tim Donkers, Benedikt Loepp, and Jürgen Ziegler. 2017. Sequential User-based Recurrent Neural Network Recommendations. In *RecSys*. 152–160.

[13] Elad Eban, Mariano Schain, Alan Mackey, Ariel Gordon, Ryan Rifkin, and Gal Elidan. 2017. Scalable Learning of Non-Decomposable Objectives. In *AISTATS*. 832–840.

[14] Simen Eide and Ning Zhou. 2018. Deep Neural Network Marketplace Recommenders in Online Experiments. In *RecSys*. 387–391.

[15] Daniel Golovin, Benjamin Solnik, Subhodeep Moitra, Greg Kochanski, John Karro, and D. Sculley. 2017. Google Vizier: A Service for Black-Box Optimization. In *KDD*. 1487–1495.

[16] Malay Haldar, Mustafa Abdool, Prashant Ramanathan, Tao Xu, Shulin Yang, Huizhong Duan, Qing Zhang, Nick Barrow-Williams, Bradley C. Turnbull, Brendan M. Collins, and Thomas Legrand. 2018. Applying Deep Learning To Airbnb Search. *CoRR* abs/1810.09591 (2018). arXiv:1810.09591 http://arxiv.org/abs/1810.09591

[17] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. 770–778.

[18] Sergey Ioffe and Christian Szegedy. 2015. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *ICML*. 448–456.

[19] Tor Lattimore and Csaba Szepesvári. 2020. *Bandit algorithms*. Cambridge University Press.

[20] Pan Li, Zhen Qin, Xuanhui Wang, and Donald Metzler. 2019. Combining Decision Trees and Neural Networks for Learning-to-Rank in Personal Search. In *KDD*. 2032–2040.

[21] Jianxun Lian, Xiaohuan Zhou, Fuzheng Zhang, Zhongxia Chen, Xing Xie, and Guangzhong Sun. 2018. xdeepfm: Combining explicit and implicit feature interactions for recommender systems. In *KDD*. 1754–1763.

[22] David C. Liu, Stephanie Rogers, Raymond Shiau, Dmitry Kislyuk, Kevin C. Ma, Zhigang Zhong, Jenny Liu, and Yushi Jing. 2017. Related Pins at Pinterest: The Evolution of a Real-World Recommender System. In *WWW*. 583–592.

[23] Tie-Yan Liu. 2009. Learning to Rank for Information Retrieval. *Found. Trends Inf. Retr.* 3, 3 (March 2009), 225–331.

[24] Rama Kumar Pasumarthi, Xuanhui Wang, Cheng Li, Sebastian Bruch, Michael Bendersky, Marc Najork, Jan Pfeifer, Nadav Golbandi, Rohan Anil, and Stephan Wolf. 2019. TF-Ranking: Scalable TensorFlow Library for Learning-to-Rank. In *KDD*.

[25] Zhen Qin, Suming J. Chen, Donald Metzler, Yongwoo Noh, Jingzheng Qin, and Xuanhui Wang. 2020. Attribute-based Propensity for Unbiased Learning in Recommender Systems: Algorithm and Case Studies. In *KDD*.

[26] Tobias Schnabel, Adith Swaminathan, Ashudeep Singh, Navin Chandak, and Thorsten Joachims. 2016. Recommendations as Treatments: Debiasing Learning and Evaluation. In *ICML*. 1670–1679.

[27] Sandeep Tata, Vlad Panait, Suming J. Chen, and Mike Colagrosso. 2019. ItemSuggest: A Data Management Platform for Machine Learned Ranking Services. In *CIDR*.

[28] Sandeep Tata, Alexandrin Popescul, Marc Najork, Mike Colagrosso, Julian Gibbons, Alan Green, Alexandre Mah, Michael Smith, Divanshu Garg, Cayden Meyer, and Reuben Kan. 2017. Quick Access: Building a Smart Experience for Google Drive. In *KDD*. 1643–1651.

[29] Ruoxi Wang, Bin Fu, Gang Fu, and Mingliang Wang. 2017. Deep & cross network for ad click predictions. In *ADKDD*. 1–7.

[30] Dawei Yin, Yuening Hu, Jiliang Tang, Tim Daly Jr., Mianwei Zhou, Hua Ouyang, Jianhui Chen, Changsung Kang, Hongbo Deng, Chikashi Nobata, Jean-Marc Langlois, and Yi Chang. 2016. Ranking Relevance in Yahoo Search. In *KDD*. 323–332.

[31] Shuai Zhang, Lina Yao, Aixin Sun, and Yi Tay. 2019. Deep Learning Based Recommender System: A Survey and New Perspectives. *ACM Comput. Surv.* 52, 1 (Feb. 2019), 5:1–5:38.

[32] Zhe Zhao, Lichan Hong, Li Wei, Jilin Chen, Aniruddh Nath, Shawn Andrews, Aditee Kumthekar, Maheswaran Sathiamoorthy, Xinyang Yi, and Ed Chi. 2019. Recommending what video to watch next: a multitask ranking system. In *RecSys*. ACM, 43–51.