2018 IEEE 34th International Conference on Data Engineering

Recommendations for All : Solving Thousands of Recommendation Problems Daily

Bhargav Kanagal #1, Sandeep Tata *2

Google, USA
1 bhargav@google.com
2 tata@google.com

***Abstract*—Recommender systems are a key technology for many online services including e-commerce, movies, music, and news. Online retailers use product recommender systems to help users discover items that they may like. However, building a large-scale product recommender system is a challenging task. The problems of sparsity and cold-start are much more pronounced in this domain. Large online retailers have used good recommendations to drive user engagement and improve revenue, but the complexity involved is a roadblock to widespread adoption by smaller retailers.**

In this paper, we tackle the problem of generating product recommendations for tens of thousands of online retailers. *Sigmund* is an industrial-scale system for providing recommendations as a service. Sigmund was deployed to production in early 2014 and has been serving retailers every day. We describe the design choices that we made in order to train accurate *matrix factorization* models at minimal cost. We also share the lessons we learned from this experience – both from a machine learning perspective and a systems perspective. We hope that these lessons are useful for building future machine-learning services.

I. Introduction

Recommendations are an important part of several online experiences. Product recommendations have been used to great effect by several large online retailers. Well designed recommender systems can help the user both before and after the purchase decision is made. For instance, by appropriately surfacing substitutes for a product being considered, recommendations can help the user make a better decision. After the decision is made (say, the item is added to the shopping cart), one can recommend suitable accessories and complements. Figure 1 shows an example of both cases.

Building a good product recommender system is a difficult task. Several challenges have been described in the literature [1], [2]. The problems of sparsity and cold-start are much more pronounced in the product recommendation domain. Product catalogs, even for a moderately sized retailer, contain several million diverse items. In the popularly researched movie domain a poor recommendation is merely a movie or a TV-show that users are likely to ignore. Arguably, a bad product recommendation carries a steeper penalty – recommending auto parts to a user who does not own a car would harm the user experience more than recommending an action movie to someone who usually watches romantic comedies. Furthermore, a growing retailer needs to perform well for new users and new products which tend to be the hardest because of the lack of data. A retailer may only know about a small number of purchases for a given user, as opposed to tens (or even hundreds) of preference ratings for movies gathered over time.

Fig. 1. Sample Recommendations generated for a user interested in a Nexus 5X phone before and after making a purchase decision.

While latent-factor based recommender systems have performed really well in contests [3], they are generally considered difficult to build, deploy, and maintain [1]. Large retailers like Amazon can marshal a sophisticated team of engineers to design and deploy a recommender system that does well for such a setting. This is often too expensive for a typical online retailer.

In this paper, we describe Sigmund, a system that allows us to solve instances of recommendation problems as a service. This system has been in production since early 2014 and serves tens of thousands of retailers daily by serving product recommendations. The system allows these retailers access to sophisticated product recommendations without having to design and deploy a recommender system. The objective of Sigmund is to provide the best possible recommendations at a low cost to thousands of retailers using a self-serve infrastructure. This raises several interesting challenges.

First, a service that processes sensitive user data such as purchases, clicks, and views of products needs to provide strong privacy guarantees. In particular, retailers want to know that their data will not be used to improve the recommenda-

2375-026X/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDE.2018.00159

IEEE computer society

tions for competitors. Sigmund guarantees this by completely separating the data and models for each of the retailers, and treating them as entirely separate instances of recommendation problems.

Second, we need to deal with the *heterogeneity* of the retailers in the modeling. In Sigmund, we have retailers that range from hundreds of items in the catalog all the way to retailers with tens of millions of items. As anyone who has deployed a recommender system knows, getting good quality recommendations in production often requires carefully tuning several hyper-parameters (e.g., number of factors) to fit the data characteristics. We describe how Sigmund automates this process in a cost-effective manner, and discuss how recommendation quality is monitored and maintained in Section III.

Third, to deal with the extreme sparsity in this domain, we need to make use of side information available about the items like product taxonomies, item brands, and item prices. We use a combination of several algorithms proposed in the recommendations literature [4], [5], [6] to compute high-quality product recommendations.

Fourth, the training and inference pipelines for Sigmund need to address the heterogeneity in the size of the retailer. Generating recommendations for smaller retailers is fairly fast, while it is takes much longer for the bigger retailers. We need to carefully design our parallelization strategy so as to handle the skew in the retailer sizes and minimize the overall makespan at low cost. We describe several techniques adopted to deal with this in Section IV.

Fifth, over 10s of thousands of retailers use Sigmund to generate recommendations on a daily basis. To keep the computational costs small, we use inexpensive pre-emptible resources (like Amazon spot instances) and carefully manage fault-tolerance. Further, we design our system to do as much of the computation as possible offline and have very lightweight computation at serving-time.

Finally, this is a continuous service – new data arrives every day, new products are introduced, and new users start shopping at these retailers. Also, new retailers sign up for the service each day. To seamlessly address this scenario, Sigmund needs to be designed to be easily *manageable* – we need to be able to deploy all tasks easily, understand and debug problems efficiently.

Note that this paper does not try to advance the state of the art in solving a single instance of a recommendation problem. In fact, approaches that leverage additional signals like product descriptions and review texts [7], [8] may offer better recommendations for items where we have little view/click/purchase data. Our contribution in this paper is to highlight the alternatives we considered for various design decisions and describe the rationale for our choices. Our choices were guided by the constraints of trying to provide the best possible recommendations for thousands of retailers at a very low cost with a small engineering team. We hope this is helpful in inspiring future academic research in this area and informative for practitioners building large-scale services using machine-learning.

II. Background

A. Data Flow

Sigmund uses views, clicks, and purchases associated with each user to train various recommendation models. In addition to this information a retailer may provide additional metadata about each of their products (images, description, attributes, etc.). The user-interaction data and product attribute data are the two main inputs to the Sigmund pipeline.

Sigmund trains different models for substitute recommendation (for use before the purchase decision) and accessory/complement recommendation (for use after the purchase decision). Once training is complete, an offline inference process materializes the recommendations for each item and retailer (using cheaper pre-emptible resources) in order to offset consuming more expensive CPU cycles at serving time. The recommendations are loaded into a distributed serving system that leverages main-memory and flash to serve low-latency requests for recommendations given a user and the associated context. Similar trade-offs have been made in previous industrial systems [2].

B. Infrastructure

The Sigmund pipeline leverages several pieces of Google infrastructure including the Google File System [9] for distributed fault-tolerant storage, and Map-Reduce [10] for scalable data processing pipelines. Section IV assumes basic familiarity with the notion of Map and Reduce tasks, and how data is accessed from a shared distributed filesystem. The features of MapReduce we describe and exploit have been discussed in the literature [10] and many are present in open-source implementations like Hadoop.

Another critical piece of underlying infrastructure is Borg [11], Google's global cluster management infrastructure. Borg provides a specification language where a service like Sigmund may describe the resources required for each of its jobs. Similar to the public cloud offerings from Google [12] and Amazon [13], it is often substantially cheaper to run offline computations such as training and inference using pre-emptible resources. In order to increase utilization, modern cluster management systems offer up unused resources at a substantial discount to regular VMs (Virtual Machines) with the caveat that these VMs can be torn down (pre-empted) with a much higher probability. When new requests arrive, the cluster management algorithm may schedule a regular VM by pre-empting low-priority VMs on a shared machine. The cost advantage of this approach over using regular VMs can be nearly 70%. However, one needs to carefully consider the overheads from fault-tolerance and recovery mechanisms to understand if the application indeed benefits from using pre-emptible resources. Section IV goes into details about how we made these choices for Sigmund.

III. Modeling Choices

Due to the extreme data sparsity in our training data, we choose a *factorization*-based model, which has been shown to

work effectively [14]. As described in Section I, we train a separate factorization model for each retailer.

A. Training Data

For each retailer, we use the traditional user-item matrix as training data. An entry in the matrix denotes an interaction between a user and an item. We consider user interactions of increasing strengths: `view` < `search` < `cart` < `conversion`. Product views have the lowest strength. A search event leading to a product view indicates more explicit intent, and therefore carries more importance. Similarly, a cart event indicates more strength than a search event, and a conversion event indicates that user bought the product. As expected, the number of conversions and cart events is orders of magnitude fewer than views and searches. Note that all of this feedback is *implicit*, we do not receive any explicit rating information from users.

B. Recommendation Model

Traditional matrix factorization algorithms e.g., Koren et al. [14] that won the Netflix challenge, do not work for implicit feedback data since we cannot contrast items liked by the user from the irrelevant items (since all known entries are ones). The only information we have is from user interaction which could indicate a like or a dislike. Two distinct approaches have been presented in the literature to handle implicit feedback: (1) pairwise ranking-based algorithms such as Rendle et al. [6] that distinguish items of positive intent (items bought, viewed, searched, etc.) from carefully sampled negative items, and (2) weighted SVD-based algorithms such as Hu et al. [15] that work on the complete matrix by exploiting structure (all unobserved entries are negative). In Sigmund, we choose BPR [6], a pairwise ranking model mainly due to its ease of implementation and its ability to model side features.

1) BPR: Pairwise Ranking Model: In BPR, a training example consists of a triple (u, i, j) user with embedding u, a positive item i with embedding v_i (this is an item that the user has previously interacted with), and a (sampled) negative item j with embedding v_j. (For convenience we use u to denote both the user and the embedding for the user.) The affinity x_{ui} between a user u and an item i is defined as the dot product between the user and the item embeddings.

$$x_{ui} = \langle u, v_i \rangle$$

The BPR model learns that the user's affinity toward the positive item is more than the user's affinity toward the negative item, i.e., $x_{ui} > x_{uj}$. The embeddings are learned such that the likelihood function below is maximized. (The Gaussian priors on the user and item embeddings are omitted for clarity.)

$$\text{argmax} \prod_{u \in U} \prod_{i \in I} \prod_{j \in I \setminus \{i\}} \sigma(x_{ui} - x_{uj})$$

Here, U denotes the set of users and I denotes the set of items. Essentially, the logistic function $\sigma(z) = \frac{1}{1+exp(-z)}$ is used to approximate the non-continuous, non-differential function: $x_{ui} > x_{uj}$.

In Sigmund, we use BPR to impose the constraint that items interacted are more important to the user than the items that are not interacted. For each interacted item, we select an unseen item as a negative (Section III-B3). Additionally, we enforce that items searched be more important than views. For every searched item, we sample a negative item that is viewed but not searched. Similarly, we also impose constraints for `cart` > `search` and `convert` > `cart`. Training a BPR model is simple with *stochastic gradient descent*. For a training example (u, i, j), we update the embeddings corresponding to u, i and j based on the loss for this example. Following the update step, the loss is guaranteed to be strictly smaller for the example. For a detailed overview of the update rules and other aspects of BPR, please refer to Rendle et al. [6]. In Sigmund, we implemented a single-machine, multi-threaded version of the update algorithm (Details in Section IV).

2) User Context: To deal with the *cold start* problem and generalize to new users, we choose to not represent users by their identifiers, but instead represent users using the *history of actions* performed. We refer to this as the *user context* in the rest of the paper. An example of a user context is (`view`: Nexus 5X, `search`: iPhone 6, `cart`: Nexus 6P)), which indicates that the user viewed a "Nexus 5X", searched for an "iPhone 6", and subsequently added a "Nexus 6P" to cart. We do not generate an explicit embedding for user u, but represent the embedding u using a linear combination of item embeddings in the context. This model allows us to generalize to new users without having to re-train the model, which is useful in complex production environments. We maintain the sequence of the past K user actions (usually about 25).

Formally, the user embedding u is computed as indicated below. Suppose that the user context is given by the sequence (I_1, I_2, …, I_K), where I_j is the (action, item pair) at the j^{th} previous step.

$$u = \sum_{j=1}^{K} w_j v^C_{I_j} \tag{1}$$

As indicated in Equation 1, we determine the user embedding using the corresponding items in the context. Note that we use a separate embedding v^C (context embedding for the item) and not the item embedding v. Also, we use weight w_j to decay the effect of user actions that are in the past.

In Figure 2, we illustrate our methodology of constructing training examples based on the user context. Here, we have four time steps and the user performs four `view` actions. The user context after the first time step is given by (a). Similarly, the user context after the second time step is (a, b). At the end of t_1, the user chooses to view item b. Suppose we sample the negative item to be item c. In this case, the BPR training example corresponds to the triple: $((a), b, c)$. Similarly, at the end of time t_2, a training example is $((a, b), c, d)$. Note that using Equation 1, we can compute the user embedding at time t_2 using the linear combination of v^C_a and v^C_b.

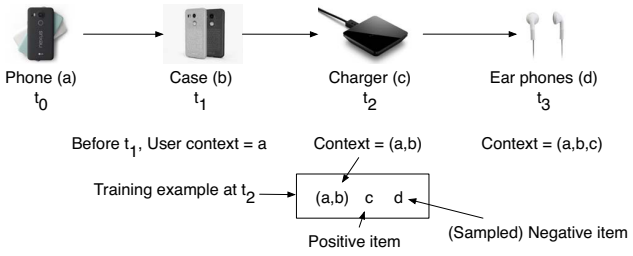

Fig. 2. Illustrating user context and how training examples are constructed.

3) Negative item sampling: The BPR model is sensitive to the choice of negative items that are sampled during the update step. We use a combination of several heuristics to sample good negative items:

- Use the item taxonomy [4] to choose items that are far away in the taxonomy (in terms of tree distance, Section III-D1).
- Exclude items that are highly co-bought/co-viewed items from negative item list.
- Sample negative items based on their affinity values as in Rendle et al. [16].

4) Features: To combat sparsity of the user-item matrix, we incorporate available auxiliary features into the model. The BPR model allows us to easily augment features on both the user and item side. Since our problem is to rank-order items, we use auxiliary features corresponding to item attributes. The list of features we use are given below.

1) *Item taxonomy:* We use the item taxonomy to generalize the item embeddings across other items that belong to the same category. For example, the item embedding for an "iPhone 6" needs to be similar to the embedding for an "iPhone 6s", and for the upcoming "iPhone 7s". Item taxonomies also help in dealing with new (cold) items. We use a hierarchical additive model similar to the one outlined in Kanagal et al. [4].
2) *Item brands & prices*: Most online shoppers are either brand-aware, i.e., they show affinity to a particular brand, or are price-conscious, i.e., they have a limited budget for a given purchase. To exploit this insight, we add the item's brands and prices as features to the BPR model (Ahmed et al. [5]). Combining this with the item taxonomy enables us to smooth *spendiness* and *brand-awareness* over the taxonomy.

C. Training & Model Selection

Training a BPR model involves determining values for the item embeddings v for each item, context embeddings v^C for each item, and corresponding parameters for the auxiliary features - taxonomy, brands and prices. One of the key challenges while training a recommendation model is to select the right features and hyper-parameters for the model. This includes selecting the right number of factors, learning rate, regularization parameters etc., all of which are critical to the performance of the model. In fact, in our experiments, we found that a model with randomly chosen hyper-parameters can be a hundred times worse (on hold-out metrics) than the best model.

In Sigmund, we need to determine the best hyper-parameters for each of the retailers in the model. The best hyper-parameters vary across retailers since they have very different data characteristics – in terms of inventory size, number of users visiting the retailer, their visitation patterns, etc. The largest retailer in our system has tens of millions items (and users), whereas the smallest retailer only has a few dozen items in the inventory. The data characteristics are also different in terms of the feature coverage. For instance, item category and brand features is missing for many small retailers. In many retailers, we found the brand coverage to be less than 10%, which makes it detrimental to add it in as a feature [17]. This means that we also need to do feature-selection separately for each retailer.

1) Grid Search: We design a scalable grid search for each of the retailers in the system in order to figure out the best possible hyper-parameter combination. For each retailer, we experiment with a range of values for each of the hyper-parameters:

- F: number of factors – To account for the wide range of retailer sizes (number of items), we experiment between 5 to 200 dimensions.
- Regularization parameters – We use separate regularizations for the item factor λ_V and for the context factor λ_{V^C}.
- Feature switches – use_brand to determine if we need to use the brand attribute as a feature, use_price, $use\_taxonomy$, etc.
- In addition, we experiment with initialization seed for RNGs, values for the prior variance, etc.

We construct a cross-product of all the parameters above and train a separate model for each of the resulting configurations (we typically restrict to around a hundred for each retailer), and select the best configuration (by hold-out metrics) for each retailer. Naturally, the grid search is expensive and consumes a significant amount of resources in our system. We discuss our approach to make this more efficient with incremental training in Section III-C3.

To set learning rates, we use the well known Adagrad [18] algorithm in conjunction with stochastic gradient descent (SGD). The Adagrad algorithm damps the learning rates of frequently updated items, and relatively increases the rate for the rare items. It works by keeping around, for each parameter, the sum of the norms of its updates. Empirically we found that Adagrad converges faster and is more reliable than the basic SGD, even for non-convex problems [19].

Bayesian methods to automatically tune hyper-parameters have been proposed in recent literature [20]; although we note that this would complicate our serving stack (having to maintain several models) and increase serving cost.

Services like Vizier [21] hold promise to improve on simple grid-search based techniques for black-box hyperparameter

optimization – both for managing trials more easily and for finding better models. If we were to rebuild the hyperparameter search today, we would design it to integrate deeply with such a service to manage the various trials. The design of such an integration is beyond the scope of this paper.

2) Goodness Metrics: For every user with more than 2 interactions, we hold out the last item in the sequence from the training data, and construct a hold out set. Note that this is a separate dataset for each retailer. In general, the higher the predicted position of the held-out item in the ranked list, the better the model's performance. There are several metrics that have been proposed for ranking-based objectives, these include AUC [22], Precision/Recall@K, n-DCG [23], and MAP (mean average precision) [22], etc.

In Sigmund, we use the MAP metric since it assigns more importance to items at the top of the ranked list. Most recommender applications are constrained to show fewer than 10 items to the user, so we use MAP@10 to evaluate our models. The naive method of computing MAP (or any other metric) requires us to compute the ranks of all the held-out items. This involves making one pass over the entire list of items (for every example in the hold-out data set), which is expensive for very large merchants. To save CPU cost, we sample 10% of the items and only estimate the MAP. We verified that this approximation does not hurt our model selection criterion. Note that we do not need to approximate MAP for small retailers.

We disregard AUC since it considers all positions on the ranked list with equal importance, which is problematic for our application. Also, for large merchants, the magnitude of the AUC difference between a good model and a mediocre one is very small (often in the fourth or fifth significant digit) and difficult to interpret.

3) Incremental training: To ensure the recommendations for the users are fresh, we need to retrain the models periodically for each of the retailers. Often, retailers add new items to the catalog, modify the sale prices on items, etc. Further, items may run out of stock. For best results, we found that we needed to refresh our models on a daily basis. Building thousands of models daily is expensive since it consumes a lot of CPU and memory. We therefore developed an incremental training strategy to save computational resources. The idea is to store the models from the previous day and continue training from there instead of starting from scratch. The embeddings for the existing items are copied over and new items are initialized with random embeddings. Note that in the incremental runs, we do not perform a complete grid search for a retailer, instead we only train the top-K most promising models (usually 3-5) from the previous day. We also note that incremental runs require much fewer iterations to converge [24]. To ensure that the incremental runs work well with Adagrad, we reset all the stored norms to 0 before the incremental update.

Periodically we restart the full model selection for all retailers in order to guarantee that the models only reflect recent history (instead of long-term history), which is a constraint imposed by the terms-of-service. Furthermore, this periodic restart helps us deal better with churn in the product inventory and locate better hyper-parameters if there is a significant change in the retailer data.

D. Inference

Inference is the problem of ranking all possible items for a given user context. A naive method is to compute affinity scores w.r.t every item and select the top-K items by score. However, this does not scale to retailers that have several millions of items. We therefore employ heuristics below, to select a subset of likely candidates (about a thousand) to recommend for each context, and only rank these items.

1) Candidate selection: We distinguish two types of recommendation tasks, given a context (shown below), and develop candidate selection strategies for each of them.

1) *View-based:* Here, we need to make a recommendation given that the user has only viewed the item page, but has not made any purchases. It is useful to recommend items that are very similar to the item (also called *substitutes* in recent literature [7]).
2) *Purchase-based:* Here, we need to make a recommendation given that the user has already purchased the item. It is useful to recommend related items such as accessories (or *complements* as described in the literature [7]).

Our candidate selection heuristics are based on a combination of item taxonomy, co-occurring items, re-purchasability, item brands and other item-specific facets such as color (e.g., for apparel), weight (e.g., for laptops), etc.

Candidates from taxonomy: A product taxonomy is a tree of categories that describe product families. A given item has ancestors that extend to the root. See Figure 3 for an example. We use the least common ancestor distance (LCA) on the taxonomy as a notion of distance between items. For instance, in Figure 3, the lca distance between the items Nexus 5X and Nexus 6P is 1. Similarly the distance between Nexus 5X and iPhone 6 is 2. Denote $lca_k(i)$ as the set of items that are at an LCA distance at most k from item i. Given an item e.g., an LG Nexus 5X, we can choose as candidates, items at lca_1, i.e., other Android phones, or we can return items at lca_2, i.e., all smart phones in general or use lca_3, i.e., all cell phones.

Candidates from co-occurrence: We use items that are co-viewed and co-bought along with the query item as candidates. $cv(i)$ denotes items that are co-viewed with item i, and $cb(i)$ denotes items co-bought with item i.

Suppose the context is given by a single item i. For view-based recommendation, we choose the set C, given by the following equation:

$$C = \cup_{j \in cv(i)} lca_k(j)$$

Essentially, we look at all the co-viewed items and look at other items that are similar to it in terms of the taxonomy. Using a small value of k keeps the recommendations precise, but will decrease coverage for tail items. On the other hand, using a large value of k provides a larger coverage at the risk

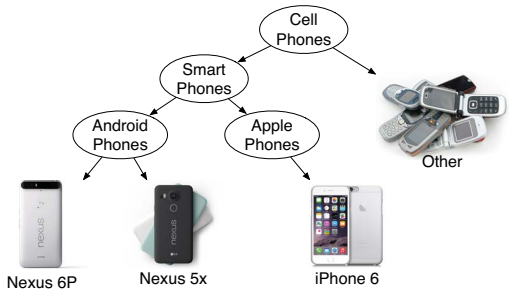

Fig. 3. Illustrating LCA: Least Common Ancestor distance. distance(Nexus 5X, Nexus 6P) = 1, distance(Nexus 5X, iPhone 6) = 2 and distance(Nexus 5X, other) = 3.

of quality. Empirically we found that setting $k = 2$ provides a good trade-off between quality and coverage.

For purchase-based recommendation we replace $cv(i)$ with $cb(i)$ in the above equation and in addition, remove items that are substitutes of i. For this setting, we empirically found that expanding with lca_1 provides the best recommendations for this case.

$$C = \cup_{j \in cb(i)} lca_1(j) \setminus lca_1(i)$$

Re-purchasing: Certain categories, e.g., diapers, water are generally repurchased. We estimate these re-purchasable categories by counting the number of users that repeat purchases from the same category. For such categories, we do not apply the set difference above. Instead we estimate the average time between purchases for such categories, and make periodic recommendations. In practice, we also distinguish between *early funnel* and *late funnel* users. For late funnel users, we focus very close to the viewed item, i.e., we select candidates that are further constrained to have the same item facets.

E. Co-occurrence models

Item-item collaborative filtering methods and their variants based on PMI (pointwise mutual information) have been successfully used in the industry [1], [25], [2] to recommend products on Amazon, movies on Netflix, and videos on YouTube. These methods are simple, general, and very scalable. They enable instantly updating recommendations, and are efficient to recompute. Empirically we found that the best way to combine the co-occurrence models along with factorization is to use the co-occurrence model for the popular items (for which we have more data) and augment the recommendations for the tail items (more sparse) from factorization. This lets us combine the best aspects of both recommenders.

IV. Systems Choices

In this section, we describe the design of the training and inference pipelines in Sigmund. We assume basic familiarity with Map-Reduce infrastructure [10] – most of the discussion is relevant to open-source implementations like Hadoop. A basic assumption in Sigmund is that a single model fits in the memory of a single machine. Since it is not uncommon to have machines with more than 128GB of memory as of the writing of this paper, this is not a serious constraint for all but the very largest retailers. Such large retailers typically build their own recommendation systems, and therefore are not the target customers for Sigmund.

We focus on the choices we made to keep the computational costs small and the pipeline *manageable*. While we run on Google's internal clusters, much like the public cloud offerings [12], [13], it is often substantially cheaper to run offline computations such as training and inference using pre-emptible resources. However, one needs to carefully consider the overheads from fault-tolerance and recovery mechanisms to understand if the application indeed benefits from using pre-emptible resources.

A. Model Selection

Choosing the right configuration of hyper-parameters for each retailer is one of the critical steps in Sigmund. The training infrastructure is set up to sweep over a specified set of combinations of hyper-parameters. A *full sweep* training run kicks off training for every combination of hyper-parameters for every retailer. This is only required when starting up Sigmund for the very first time, or when restarting the training and inference infrastructure after a catastrophic loss of all the models. An *incremental sweep*, as the name suggests, only trains a small set of models (typically 3) for each retailer corresponding to the best performing combinations of hyper-parameters. The models are ordered by MAP@10 on a hold-out set. Further, incremental sweep uses the models trained in the previous run to initialize the parameters, and therefore often converges faster than random initialization. Since starting the Sigmund service, we have only had to resort to a full-sweep after planned temporary turn-downs in the incremental sweep pipeline. An incremental sweep may include a new retailer that has signed up for the service, in which case Sigmund trains all possible combinations of hyper-parameters for that retailer alone. The sweep step determines the overall set of models to train, and outputs a set of *config records* containing the model number, training and validation dataset locations, and the values assigned to each of the hyperparameters. These config records form the input to the training step.

B. Training

Conceptually, the design of the training job is simple. The input collection of config records specifies the list of models that need to be trained – we simply need to call a *Train()* function on each input. This function reads the data at the location specified, trains a model, and writes it out to the specified location. After training, *Train()* also evaluates the learned model on a specified hold-out set and writes out the goodness metrics (Section III-C2) in an *output config record*. Training is implemented as a MapReduce job where the map phase executes training and evaluation, and the reduce phase writes out the output config records to disk. The main challenge in designing the training pipeline comes from having to balance efficiency concerns with manageability. Figure 4

shows a schematic of how Sigmund structures the training process.

1) Low-Cost Resources: In order to execute the training job at low cost, we schedule the map tasks in the MapReduce as low priority tasks that are pre-emptible. Fault-tolerance approaches for dealing with pre-emptions are described in Section IV-B3. We identify data centers that have unused resources, and break down the job into several independent MapReduces so that there is one for each data center. Since training using SGD iterates over the data multiple times, we simply migrate the training data to the data center where the computation is run. The cost of training is dominated by the CPU cost of making SGD steps, and the network cost of moving the data usually ends up producing a net benefit.

The input config records are randomly permuted before being written so that training tasks are randomly divided across different MapReduces. We also rely on this randomization strategy to balance the work within a MapReduce job. Workers assigned small retailers process more training tasks, and those with larger retailers process fewer training tasks in a single job.

2) Multi-threading: Using multiple threads effectively can greatly reduce the cost of training. The trivial way to take advantage of multi-core processors is to schedule several map tasks on the same physical machine. Each map task on a machine processes a separate chunk of the input, independent of the processing in other tasks. These tasks can either be scheduled as separate processes or as separate threads within a process to potentially take advantage of shared data structures (such as dictionaries, codecs, etc.). Google's MapReduce implementation supports this and several other options for taking advantage of modern multi-core processors. However, this is not the most efficient way to use the resources at hand for training. Recall that we make the assumption that a single retailer's model fits in the memory of a single machine. The trivial approach of scheduling multiple map tasks on a machine may end up scheduling models from more than one retailer on the same machine. While this may work for smaller retailers, scheduling two large retailers on the same machine could exceed the available memory. Specifying this scheduling constraint is difficult since we need a mechanism to estimate the memory footprint of each task by inspecting the contents of the config records within the chunk of input assigned to it. This requires user-code to be executed by the scheduler, and can be brittle in the face of evolving code, and prevents many advanced MapReduce features like dynamic resizing of input chunks for better load-balancing.

Instead of implementing a complex and brittle scheduling constraint, we chose to train only a single retailer on a physical machine at a time, and instead use multiple threads to train faster. We use Hogwild-style multi-threaded training [26]. Instead of relying on the MapReduce framework to manage multi-threading, we train each model corresponding to each input config record using multiple threads that are managed in the user code. We rely on a dynamically sized virtual machine to use small amounts of memory for tasks that are training small retailers, and large amounts of memory for tasks with larger retailers.

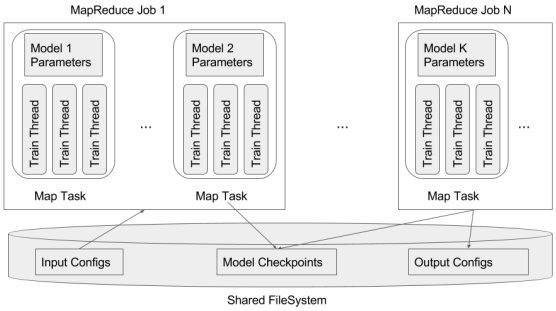

Fig. 4. Training Schematic

Note that training requires allocating memory for all the model parameters that are going to be learned. This takes up the same amount of memory whether we run a single training thread or multiple threads. Once we have allocated the memory, we want to make the most of it, and requesting CPUs to run additional training threads helps us make more efficient use of the memory already requested.

Much like with public cloud machine types, high-memory instances tend to be correlated with high CPU. For instance, it is often more cost-effective to get four CPUs and 32GB rather than one CPU with 32GB for the training job.

3) Fault-tolerance: A direct consequence of using cheap pre-emptible VMs is that the design needs to consider pre-emptions, failures and machine restarts. During training, we asynchronously checkpoint the model learned to a shared filesystem. We use the strategy of scheduling checkpoints on a fixed time-interval (e.g., every few minutes) instead of scheduling them after a fixed number of iterations. This choice was motivated by the heterogeneity of the retailers in terms of size and complexity – from small retailers with a few thousand user-item interactions to large retailers with hundreds of millions of interactions (time per iteration across retailers varies significantly). This approach gives us a way to control the amount of work lost on pre-emption. Furthermore, we only need to keep the latest checkpoint around, so as soon as a new checkpoint is written, we garbage-collect the previous checkpoint. The checkpointing itself is very fast and is negligible compared to the training time.

C. Inference

We run an offline inference job to materialize item-item recommendations. This enables us to keep the computational cost of serving low, and deal quickly with new users without having to retrain the model. Inference proceeds by examining the output config records emitted by the training job to identify the best model for each retailer. Conceptually, for each retailer, the job considers each item in the inventory, and produces the corresponding top ranked items predicted by the model. Like the training job, this is also implemented using MapReduce.

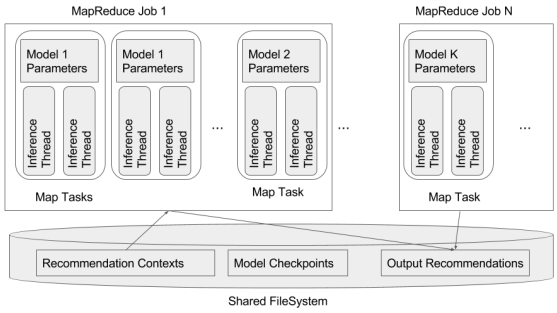

Fig. 5. Inference Schematic

The inference step is done in the map phase. The reduce phase writes out the computed recommendations for each item. Figure 5 shows a schematic of how the inference pipeline is organized.

The input consists of the union of all the items from each retailer. Item IDs contain the retailer ID, so the same item sold by different retailers will have a different ID.

1) Parallelization: Once again, we split the input into several distinct sets so that the computation can take advantage of resources in multiple cells. This is the same principle as in the training job, however, there is an important difference in how the computation is parallelized. While training is parallelized in a way that limits the computation of a given model for a given retailer to be executed on a single machine, this is not the case for inference. For a large retailer with many items, inference may be parallelized over hundreds of machines, while for a small retailer, it may happen on a single machine.

Inference is complete when the MapReduces in each of the data centers finish. To minimize the total running time of the job, we use a greedy first-fit bin-packing heuristic to partition the retailers. The computational cost of inference is roughly linearly proportional to the number of items. This is because the candidate selection logic(Section III-D1) limits the number of candidates we need to consider for each item. We therefore use the number of items in each retailer's inventory as the weight for that retailer in the bin-packing problem. In contrast, a naive approach that computed the affinity for every pair of items would use the square of the number of items in the retailer's inventory.

2) Multi-threading: Similar to the training job, we manage multi-threading in the application code. MapReduce processes one record at a time, which in the case of inference is an item record. Since we produce a union of all the input data, the next record could come from a different retailer. We organize the input data in such a way that data from a single retailer is in one contiguous chunk. The map function loads the model for the current retailer into memory (if it is not already there). A load should only get triggered if this is the first record being processed by the mapper or if it is processing an input split that contains the boundary between two retailers. We configure MapReduce to run a single map thread in a task so that it doesn't have to load more than one model into memory at once. Without this restriction, it is likely that a single machine may process two different large retailers and run out of available main memory. However, in order to make full use of the fact that we have the model in memory, we use multiple threads to evaluate concurrently. This is accomplished by writing a multi-threaded function within the map task while the MapReduce framework is configured *not* to process multiple splits concurrently in a task.

The ideal system would allow us to use multi-threaded map tasks such that a mapper would either be assigned tasks corresponding to the same retailer or multiple retailers such that the models fit in memory. This is difficult to accomplish within the constraints of MapReduce. We decided that the complexity of testing and deploying a new task-scheduling algorithm was not worth the additional efficiency we may have been able to obtain.

V. Discussion

We considered many alternatives to the design described above to organize the computational work of model selection, inference and serving. One key choice was the use of MapReduce to structure and distribute our computation even though Sigmund is not a conventional data-parallel pipeline. While this choice implied we gave up some potential efficiencies, the gains in manageability made it worthwhile. The choice of MapReduce allowed us to quickly scale from a few hundred retailers in the program when we first started the service to tens of thousands within 18 months.

One possibility we considered was to build a custom scheduling and monitoring service that would schedule the training and inference for each retailer in a data center with free resources. Building this service in a fault-tolerant way would have taken considerable effort. Furthermore, we decided early on that the system would not require support for charge-backs, where each retailer would be charged for computational resources consumed and would therefore need to be metered separately. This allowed us to simplify the design substantially. Finally, the serving infrastructure can now be optimized for batch-updates every time we have the inference job complete for all the retailers in the system. While this constrains how often the serving stack is updated, it does away with having to design it for real-time updates for millions of items across thousands of retailers.

We do believe there are many other ways to build such a service if presented with other constraints like higher costs from the computational platform, lower bar for recommendation quality, or even the necessity to implement charge-backs to each retailer in proportion to the resources consumed to compute and serve recommendations for them.

Offline metrics do not directly translate to improvements in online metrics (e.g., conversions on recommendations). Our training data only allows us to optimize for clicks and views since the conversions are too sparse to model accurately. So,

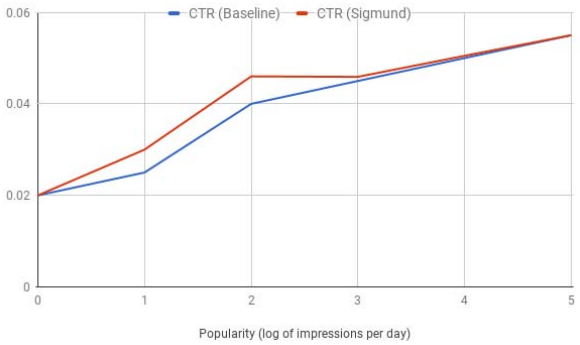

Fig. 6. Cross-retailer plot of an item's popularity in terms of impressions per day and the CTR of that item when it is showed as a recommendation

we relied on a series of carefully structured online experiments to inform our design choices as we improved the models. Some aspects that we'd have liked to model, such as the quality of the images provided by a retailer, budgets of the retailers, inventory levels, were beyond the scope of this project.

Since the model for each retailer is optimized independently, our approach treats large and small retailers fairly – producing the best possible model for the data they provide. We observe that Sigmund improves engagement heavily for the long tail of products, and has little impact on the extremely popular products. Figure 6 shows the impact of Sigmund on the long tail. The graph plots the average CTR for an item with its popularity (measured as the number of times it is shown daily) across all retailers we serve averaged over a 7-day period. The CTR values are scaled to accurately represent the relative improvements without disclosing absolute numbers. The baseline chosen for comparison is to a simple co-occurrence model. As the graph shows, Sigmund's recommendations see significantly higher engagement for less popular items (the long tail) while they have virtually no effect on highly popular items.

VI. Related Work

Recent papers [27] provide a broad overview of recommendation techniques in industry. In practical settings, implicit feedback data (click, view, etc. rather than ratings) is much cheaper to obtain than explicit feedback. Approaches based on pair-wise ranking [16], [6], [28]), and least-squares [15], [29] have been proposed in the literature. Although we chose BPR for its simplicity and extensibility with feature engineering, we can easily substitute it with the least-squares approach.

Using auxiliary features is a useful method to combat sparsity in the training data and help with the cold-start problem. Agarwal et al. [30] propose a joint regression and factorization model to include auxiliary features, Koeningstein et al. [31] model hierarchies such as movie taxonomies and the temporal nature of user behavior. Basilico and Raimond [32] argue to incorporate time as a first-class element when modeling a user's interests. In recent years, McAuley et al. [33] and Agarwal et al. [8] propose techniques to use the sentiment of user reviews to tweak recommendations. McAuley et al. [7] have used auxiliary information to identify substitutes and complements for items.

While our recommendation engine was custom-built based on matrix factorization, models that use neural networks [34] have shown significant promise. Replacing our core engine with a neural recommender using a framework like Tensorflow [35] while retaining the infrastructure to solve thousands of instances in parallel is an engineering exercise.

Recent years have seen many Machine Learning services being offered in the Cloud. Google Prediction API [36], Amazon Machine Learning [37], and Microsoft Azure [38] provide services for classification and regression and general purpose machine learning. To the best of our knowledge Sigmund is the first industrial scale product recommendation service described in the literature.

VII. Conclusions

On the modeling front, we observed from thousands of instances that co-occurrence based recommendations work well with large amounts of data; more sophisticated techniques rarely outperform it. This is congruent with claims made by other practitioners in industry [39]. Having said that, we were able to empirically demonstrate the value of matrix-factorization-style approaches for the long tail of product inventories which are increasingly important in online retail. Using co-occurrence based recommendations for the popular items, and augmenting them with factorization-derived recommendations allows us to cover a much larger fraction of the inventory with good recommendations.

When building a service like Sigmund, it is critical to design away any manual per-retailer configuration and tuning. Without this, we would not have been able to design, build, and deploy this system with a team of just two engineers working over a few months. A key early assumption we made, to use a non-distributed implementation for the solver did not prove to be a problem. Given the size of main memories today, we were able to solve instances with several million items in the product inventory.

We solved a particularly hard problem, one of model selection, through a carefully engineered and self-managed grid search over the space of hyper-parameters. While on the surface this seems expensive, we pay for this search only once. Incremental training only considers the top few models for each retailer. The choices we made to increase manageability at the cost of potentially sacrificing efficiency also turned out to be prudent.

One of the interesting opportunities for future work is to further refine the settings for which recommendations are produced and displayed. For instance, the recommendations that are most useful for a casual shopper who's trying to explore options for a couch in their living room are different from those for a user who knows they want a certain style of couch, which are in turn are different from those for a user who has determined the exact couch she wants and is looking

for matching accessories. Providing this additional level of tailoring of recommendations while keeping the overall system manageable is ongoing work.

Our choice of using a ranking objective function (like BPR) lets us optimize using a simple gradient-descent solver. This approach makes it easy to produce a ranked list of recommendations, but it is difficult to estimate the absolute relevance of the recommendation, particularly if we want to make a decision on whether to display to the user. We are considering future approaches that combine the advantages of a BPR-style ranking objective with the ability to provide a relevance score that can be compared to a threshold.

REFERENCES

[1] X. Amatriain and J. Basilico, "Netflix Recommendations: Beyond the 5 Stars," http://techblog.netflix.com/2012/04/netflix-recommendations-beyond-5-stars.html.

[2] G. Linden, B. Smith, and J. York, "Amazon.com recommendations: Item-to-item collaborative filtering," *IEEE Internet Computing*, vol. 7, no. 1, pp. 76–80, Jan. 2003. [Online]. Available: http://dx.doi.org/10.1109/MIC.2003.1167344

[3] J. Bennett, C. Elkan, B. Liu, P. Smyth, and D. Tikk, "Kdd cup and workshop 2007," *SIGKDD Explor. Newsl.*, vol. 9, no. 2, pp. 51–52, Dec. 2007. [Online]. Available: http://doi.acm.org/10.1145/1345448.1345459

[4] B. Kanagal, A. Ahmed, S. Pandey, V. Josifovski, J. Yuan, and L. Garcia-Pueyo, "Supercharging recommender systems using taxonomies for learning user purchase behavior," *PVLDB*, vol. 5, no. 10, pp. 956–967, Jun. 2012. [Online]. Available: http://dx.doi.org/10.14778/2336664.2336669

[5] A. Ahmed, B. Kanagal, S. Pandey, V. Josifovski, L. G. Pueyo, and J. Yuan, "Latent factor models with additive and hierarchically-smoothed user preferences," in *WSDM*, 2013.

[6] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "BPR: bayesian personalized ranking from implicit feedback," in *UAI*, 2009.

[7] J. McAuley, R. Pandey, and J. Leskovec, "Inferring networks of substitutable and complementary products," in *KDD*, 2015.

[8] D. Agarwal and B. Chen, "flda: matrix factorization through latent dirichlet allocation," in *WSDM*, 2010.

[9] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," *SIGOPS Oper. Syst. Rev.*, vol. 37, no. 5, pp. 29–43, Oct. 2003. [Online]. Available: http://doi.acm.org/10.1145/1165389.945450

[10] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *CACM*, vol. 51, no. 1, pp. 107–113, 2008.

[11] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *EuroSys*, 2015.

[12] Google, "Preemptible VMs," https://cloud.google.com/preemptible-vms/.

[13] Amazon, "Spot Instances," https://aws.amazon.com/ec2/spot/.

[14] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *IEEE Computer*, vol. 42, no. 8, pp. 30–37, 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.263

[15] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *ICDM*, 2008.

[16] S. Rendle and C. Freudenthaler, "Improving pairwise learning for item recommendation from implicit feedback," in *WSDM*, 2014. [Online]. Available: http://doi.acm.org/10.1145/2556195.2556248

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

[18] J. C. Duchi, E. Hazan, and Y. Singer, "Adaptive subgradient methods for online learning and stochastic optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2021068

[19] M. R. Gupta, S. Bengio, and J. Weston, "Training highly multi-class linear classifiers," *Journal Machine Learning Research*, 2014. [Online]. Available: http://jmlr.org/papers/volume15/gupta14a/gupta14a.pdf

[20] S. Rendle, "Learning recommender systems with adaptive regularization," in *WSDM*, 2012.

[21] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley, "Google vizier: A service for black-box optimization," in *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '17. New York, NY, USA: ACM, 2017, pp. 1487–1495. [Online]. Available: http://doi.acm.org/10.1145/3097983.3098043

[22] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge University Press, 2008.

[23] W. B. Croft, D. Metzler, and T. Strohman, *Search Engines - Information Retrieval in Practice*. Pearson Education, 2009.

[24] S. Bengio and G. Heigold, "Word embeddings for speech recognition," in *ISCA*, 2014.

[25] J. Davidson, B. Liebald, J. Liu, P. Nandy, T. Van Vleet, U. Gargi, S. Gupta, Y. He, M. Lambert, B. Livingston, and D. Sampath, "The youtube video recommendation system," in *RecSys*, 2010.

[26] F. Niu, B. Recht, C. Ré, and S. Wright, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *NIPS*, 2011.

[27] B. Smith and G. Linden, "Two decades of recommender systems at amazon.com," *IEEE Internet Computing*, vol. 21, no. 3, pp. 12–18, 2017.

[28] J. Weston, S. Bengio, and N. Usunier, "Large scale image annotation: Learning to rank with joint word-image embeddings," in *ECML*, 2010. [Online]. Available: http://www.kyb.mpg.de/bs/people/weston/papers/wsabie-ecml.pdf

[29] M. Weimer, A. Karatzoglou, Q. V. Le, and A. J. Smola, "COFI RANK - maximum margin matrix factorization for collaborative ranking," in *NIPS*, 2007, pp. 1593–1600.

[30] D. Agarwal and B. Chen, "Regression-based latent factor models," in *SIGKDD*, 2009.

[31] N. Koenigstein, G. Dror, and Y. Koren, "Yahoo! music recommendations: modeling music ratings with temporal dynamics and item taxonomy," in *RecSys*, 2011.

[32] J. Basilico and Y. Raimond, "Déjà vu: The importance of time and causality in recommender systems," in *Proceedings of the Eleventh ACM Conference on Recommender Systems*, ser. RecSys '17. New York, NY, USA: ACM, 2017, pp. 342–342. [Online]. Available: http://doi.acm.org/10.1145/3109859.3109922

[33] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *RecSys*, 2013.

[34] P. Covington, J. Adams, and E. Sargin, "Deep neural networks for youtube recommendations," in *Proceedings of the 10th ACM Conference on Recommender Systems*, ser. RecSys '16. New York, NY, USA: ACM, 2016, pp. 191–198. [Online]. Available: http://doi.acm.org/10.1145/2959100.2959190

[35] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[36] Google, "Google Prediction API," https://cloud.google.com/prediction/docs.

[37] Amazon, "Amazon Machine Learning," https://aws.amazon.com/machine-learning/.

[38] Microsoft, "Azure Machine Learning," https://azure.microsoft.com/en-us/services/machine-learning/.

[39] D. Agarwal, "Scaling machine learning and statistics for web applications," in *SIGKDD*, 2015, p. 1621. [Online]. Available: http://doi.acm.org/10.1145/2783258.2790452