Extreme Multi-label Learning with Label Features for Warm-start Tagging, Ranking & Recommendation

Yashoteja Prabhu[*]
yashoteja.prabhu@gmail.com

Anil Kag[†]
anilkagak2@gmail.com

Shilpa Gopinath[‡]
shilpagopitvm@gmail.com

Kunal Dahiya[*]
kunalsdahiya@gmail.com

Shrutendra Harsola[†]
shharsol@microsoft.com

Rahul Agrawal[†]
Rahul.Agrawal@microsoft.com

Manik Varma[*][†]
manik@microsoft.com

ABSTRACT

The objective in extreme multi-label learning is to build classifiers that can annotate a data point with the subset of relevant labels from an extremely large label set. Extreme classification has, thus far, only been studied in the context of predicting labels for novel test points. This paper formulates the extreme classification problem when predictions need to be made on training points with partially revealed labels. This allows the reformulation of warm-start tagging, ranking and recommendation problems as extreme multi-label learning with each item to be ranked/recommended being mapped onto a separate label. The SwiftXML algorithm is developed to tackle such warm-start applications by leveraging label features. SwiftXML improves upon the state-of-the-art tree based extreme classifiers by partitioning tree nodes using two hyperplanes learnt jointly in the label and data point feature spaces. Optimization is carried out via an alternating minimization algorithm allowing SwiftXML to efficiently scale to large problems.

Experiments on multiple benchmark tasks, including tagging on Wikipedia and item-to-item recommendation on Amazon, reveal that SwiftXML's predictions can be up to 14% more accurate as compared to leading extreme classifiers. SwiftXML also demonstrates the benefits of reformulating warm-start recommendation problems as extreme multi-label learning tasks by scaling beyond classical recommender systems and achieving prediction accuracy gains of up to 37%. Furthermore, in a live deployment for sponsored search on Bing, it was observed that SwiftXML could increase the relative click-through-rate by 10% while simultaneously reducing the bounce rate by 30%.

[*]Indian Institute of Technology Delhi
[†]Microsoft Research and AI
[‡]Samsung Research

KEYWORDS

Extreme multi-label learning, Sponsored search, Large scale recommender systems with user and item features

ACM Reference Format:
Yashoteja Prabhu, Anil Kag, Shilpa Gopinath, Kunal Dahiya, Shrutendra Harsola, Rahul Agrawal, and Manik Varma. 2018. Extreme Multi-label Learning with Label Features for Warm-start Tagging, Ranking & Recommendation. In *WSDM 2018: WSDM 2018: The Eleventh ACM International Conference on Web Search and Data Mining , February 5–9, 2018, Marina Del Rey, CA, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3159652.3159660

1 INTRODUCTION

Objective: This paper studies extreme multi-label problems where predictions need to be made on training points with partially revealed labels rather than on previously unseen test points. The SwiftXML algorithm is developed for learning in such scenarios by exploiting both label and data point features and applied to warm-start tagging, ranking and recommendation applications.

Extreme classification: Extreme multi-label learning addresses the problem of learning a classifier that can annotate a data point with the most relevant subset of labels from an extremely large label set. Note that multi-label learning is distinct from multi-class learning which aims to predict a single mutually exclusive label.

Extreme classification has opened up a new paradigm for thinking about applications such as tagging, ranking and recommendation. In general, one can reformulate these problems as extreme classification tasks by treating each item to be ranked/recommended as a separate label, learning an extreme multi-label classifier that maps a user's feature vector to a set of relevant labels, and then using the classifier to predict the subset of items that should be ranked/recommended to each user. Data points are therefore referred to as users and labels as items throughout this paper.

Unfortunately, extreme classification has thus far been studied only in the cold-start context of recommending items to new users (predicting labels for novel test points). As a result, existing extreme classification algorithms have been based on user features alone and have not exploited item features. They have therefore been unable to leverage information about the revealed item preferences of existing users available in warm-start scenarios.

Motivation and label features: This paper is motivated by the applications of tagging on Wikipedia, item-to-item recommendation on Amazon and ranking queries for a given ad-landing page for sponsored search on Bing. In the case of tagging Wikipedia articles, each Wikipedia tag is treated as a separate label and the tag's word2vec embedding is treated as its label feature vector. SwiftXML is then used to predict new tags for an existing article leveraging information about not only the article's text but also the existing tags for the article. For item-to-item recommendation on Amazon, each item is treated as a separate label and bag-of-words label features are extracted based on the product description. SwiftXML is then used to recommend other items that might be bought along with a given item leveraging not only the given item's product description but also information about existing item recommendations that had been adopted by users in the past. Finally, for sponsored search advertising, each of the top monetizable queries on Bing is treated as a separate label and label features are extracted based on CDSSM embeddings [25] for each query. SwiftXML is then used to rank the relevant queries for a given ad landing page leveraging not only the page's content but also information about existing queries that had led to clicks on the ad in the past as well as existing queries that the advertiser had bid on for that ad. This allows SwiftXML to make significantly more accurate predictions as compared to previous extreme classification algorithms which could not leverage information about existing tags, previously recommended items and relevant queries.

Limitations of current recommender systems: Warm-start recommendation is a very well studied problem and a number of algorithms have been proposed outside the extreme classification literature including those based on matrix factorization [12, 14, 16, 24], Inductive Matrix Completion [20], Local Collaborative Filtering [17], *etc.* Unfortunately, most of these algorithms cannot scale to the extreme setting as they cannot train on large item and user feature matrices involving millions of items and users in reasonable time on commodity hardware. Furthermore, some of these algorithms are unable to handle implicit ratings (where missing labels are not necessarily irrelevant) and, given millions of items, are unable to make predictions in milliseconds, both of which are critical features for deploying in real world applications. While some heuristics have been proposed to address these limitations, such as adding a tree based search over the items as a post-processing step [33], principled solutions which directly incorporate implicit ratings and logarithmic prediction costs into the learning algorithm have not been proposed for large scale recommender systems based on user and item features. Finally, many of these algorithms, with the notable exception of Local Collaborative Ranking [17], are based on the assumption that the ratings matrix is low-rank and therefore have poor prediction accuracies as this assumption is violated in the extreme setting [7].

SwiftXML: SwiftXML addresses these limitations by efficiently learning an ensemble of tree classifiers at the extreme scale. As with other extreme classification algorithms, SwiftXML can train on implicit ratings without the low-rank assumption and can make predictions in milliseconds. Unlike other extreme classification algorithms, however, SwiftXML grows its trees by recursively partitioning nodes using two hyperplanes learned jointly in the user and item feature spaces. This improves training over learning a single hyperplane based on user features alone in the following ways. First, users who were incorrectly partitioned based on user features alone might now be partitioned correctly using the additional item features. Second, users who like similar items can now be partitioned together as their item features will be similar. Predictions are made exactly as in existing tree-based extreme classifiers but for the fact that trees are now traversed based on the sign of the average prediction of both hyperplanes at a node. This improves prediction accuracy by allowing SwiftXML to recommend items that are similar to the items that are already liked by a user not just in terms of ratings similarity as in the case of existing extreme classifiers but also in terms of item feature similarity. Finally, prediction accuracy also improves over the traditional extreme classification formulation as more information is now available at test time.

Contributions: This paper extends the extreme classification formulation to handle warm-start scenarios and develops, to the best of our knowledge, the first extreme classification algorithm incorporating label features. The key technical contributions are a novel tree node splitting function based on both user and item features and a scalable algorithm for optimizing the function. Experimental results reveal that SwiftXML's predictions can be as much as 14% more accurate as compared to state-of-the-art extreme classifiers and 37% more accurate as compared to classical methods for warm-start recommendation. Another significant contribution is the live deployment of SwiftXML to real users for sponsored search on Bing. It is demonstrated that SwiftXML can increase the relative click-through-rate by 10% while simultaneously reducing the bounce rate by 30%.

2 RELATED WORK

Much progress has recently been made in developing extreme learning algorithms based on trees [5, 15, 22, 29], embeddings [7, 10, 13, 19, 27, 28, 34] and linear approaches [6, 31, 32]. While these methods might potentially be used to make warm-start recommendations, their performance degrades in this setting since they do not leverage item feature information during training and do not exploit information about a user's revealed item preferences during prediction.

Direct approaches to extreme learning, such as DiSMEC [6] learn and apply a separate linear classifier per label. As a result, they have the highest prediction accuracies but can also take months for training and prediction at extreme scales on commodity hardware. Tree-based extreme classifiers, such as PfastreXML [15] have lower prediction accuracies and larger model sizes but can train in hours on a small server and can make predictions in milliseconds per test point. Scaling to large datasets is a critical requirement for deployment in Bing and one of the major advantages that extreme classifiers enjoy over traditional warm-start recommendation algorithms based on user and item features. SwiftXML is therefore developed as an extension of PfastreXML. SwiftXML enjoys all the scaling properties of PfastreXML while having significantly higher prediction accuracies than PfastreXML, DiSMEC and state-of-the-art recommender systems.

Warm-start prediction has been well-studied in the recommender systems literature. Matrix factorization techniques such as WRMF [14], SVD++ [16], BPR [24] and WBPR [12], were specifically designed to address this scenario but factorize the ratings matrix without exploiting user or item feature information. This limitation has been addressed in IMC [20], Matchbox [26], LCR [17] and other methods [4]. Unfortunately, none of these scale to the extreme setting involving a large number of users and items with implicit ratings. Furthermore, with the exception of LCR, each of these methods assumes that the ratings matrix is low-rank, which does not hold in the extreme setting [7]. Consequently these methods have poor prediction accuracies as demonstrated in the Experiments Section. While some heuristics such as tree based post-processing search [33] have been proposed to speed-up prediction, they are not specialized for implicit rating scenarios and don't address the training time and accuracy concerns.

3 A MOTIVATING EXAMPLE

Consider the problem of tagging an existing Wikipedia article – such as that of Albert Einstein which has already been annotated with about 60 tags. State-of-the-art extreme classifiers suffer from the following three limitations. First, during training, they learn impoverished models from article text features alone and are unable to leverage features from tags such as "Recipients of the Pour le Mérite" and "Members of the Lincean Academy" which contain information not found in the article's text. SwiftXML avoids this issue by learning from word2vec features extracted from the tags in addition to the article text features. This allows SwiftXML to predict "Recipients of civil awards and decorations" for the article which could not be predicted by the traditional methods. Second, again during training, existing classifiers learn that Einstein and Newton's articles are very different as they share very few tags. However, SwiftXML learns that the two are similar as the word2vec embeddings of "American physicists" and "English physicists" as well as other corresponding tags are similar. This allows SwiftXML to annotate Einstein's article with Newton's tag "Geometers" which could not have been predicted from the article text directly. Finally, during prediction, SwiftXML can confidently annotate Einstein's article with a novel tag "Astrophysicists" by relying on its similarity to the existing tag "Cosmologists" in terms of their label features. Existing extreme classifiers are unable to leverage such label correlations.

4 SWIFTXML

This section develops the SwiftXML algorithm as an extension of PfastreXML for addressing warm start classification tasks. SwiftXML retains all the scaling properties of PfastreXML while being significantly more accurate owing to the effective use of additional information about revealed item preferences and item features. SwiftXML node partitioning optimizes a novel objective which jointly learns two separating hyperplanes in the user and item feature spaces respectively.

Classifier architecture: SwiftXML trees are grown by recursively partitioning the users into 2 child nodes. Each internal node stores 2 separating hyperplanes in the user and the item feature spaces respectively, which jointly learn the user partitioning. Since learning an optimum user partition is a computationally hard problem, SwiftXML proposes an efficient alternating minimization algorithm which converges to a locally optimal solution. The joint training allows information sharing between the 2 hyperplanes, leading to better partitions. The tree growth is terminated when all the leaf nodes contain less than a specified number of users which is a hyperparameter of the algorithm. Each leaf node stores a probability distribution over the items which are proportional to number of users in the leaf that prefer the respective items. To rectify the mistakes committed by a single, deep tree, SwiftXML trains multiple trees which differ only in the random seed values used to initialize the node partitioning algorithm. SwiftXML tree prediction involves passing the test user down each of the trees, starting from the root node until it reaches a leaf node. At each node, the test point is sent to left (right) child if the average of the scores predicted by the 2 hyperplanes is positive (negative). The probability scores of all the reached leaf nodes are averaged to obtain the label probability predictions. The SwiftXML trees are prone to predicting tail labels with low probabilities as partitioning errors in the internal nodes disproportionately reduce the support of tail labels in the leaf node distributions. SwiftXML addresses this limitation by re-ranking the tree predictions using classifiers designed specifically for tail labels. SwiftXML follows the same training and prediction procedure for tail label classifiers as [15]. SwiftXML trees were empirically observed to be well-balanced, leading to logarithmic training and prediction complexities.

Item-set features: SwiftXML nodes learn a hyperplane in the item-set feature space, jointly with a hyperplane in the user feature space. The item-set features encode semantic information about a user's revealed item preferences. The following linear formula is used for deriving the item-set features of an ith user: $\mathbf{z}_i = (\sum_j y^r_{ij}\mathbf{x}'_j)/\|\sum_j y^r_{ij}\mathbf{x}'_j\|$, where $\mathbf{y}^r_i \in \{0,1\}^L$ denotes the revealed item preference vector for the ith user, and $\mathbf{x}'_j \in \mathbb{R}^{D'}$ denotes the feature vector for the jth item. This formulation ensures that users with overlapping item preferences have similar item-set features, which helps to retain those users together in the SwiftXML trees. The item-set features provide information complementary to the user features, and allow SwiftXML to leverage correlations that exist between the revealed and the test items. To account for varying number of revealed items across users, the item-set features are normalized to unit norm.

Node partitioning objective function: SwiftXML optimizes a novel node partitioning objective which is designed to ensure both purity as well as generalizability of the learned partitions:

$$\begin{aligned}
&\textbf{Min } \|\mathbf{w}_x\|_1 + C_x \sum_i \mathcal{L}_{\text{reg}}(\delta_i \mathbf{w}_x^\top \mathbf{x}_i) + \|\mathbf{w}_z\|_1 + C_z \sum_i \mathcal{L}_{\text{reg}}(\delta_i \mathbf{w}_z^\top \mathbf{z}_i) \\
&+ C_r \sum_i \left(\frac{1+\delta_i}{2} \mathcal{L}_{\text{rank}}(\mathbf{r}^+, \mathbf{y}^r_i) + \frac{1-\delta_i}{2} \mathcal{L}_{\text{rank}}(\mathbf{r}^-, \mathbf{y}^r_i) \right) \\
&\textbf{w.r.t. } \mathbf{w}_x \in \mathcal{R}^D, \mathbf{w}_z \in \mathcal{R}^{D'}, \boldsymbol{\delta} \in \{-1,+1\}^L, \mathbf{r}^+, \mathbf{r}^- \in \Pi(1,L) \\
&\textbf{where } \mathcal{L}_{\text{reg}}(x) = \log(1+e^{-x}) \textbf{ , } \mathcal{L}_{\text{rank}}(\mathbf{r}, \mathbf{y}) = -\frac{\sum_{l=1}^{L} \frac{y_l}{p_l \log(r_l+1)}}{\sum_{l=1}^{L} \frac{1}{\log(l+1)}}
\end{aligned} \tag{1}$$

Here, i enumerates the training users; $\delta_i \in \{-1,+1\}$ indicates the user assignment to either negative (right) or positive (left) partition; $\mathbf{w}_x, \mathbf{w}_z$ represent the separating hyperplanes learned in the user and item-set feature spaces; $\mathbf{r}^+$ and $\mathbf{r}^-$ represent the item ranking variables for positive and negative partitions; $\Pi(1, L)$ denotes the space of all possible rankings over the L items; C_x, C_z, C_r are SwiftXML hyper-parameters; p_l are the item propensity scores [15].

The first line in (1) promotes model sparsity as well as generalizability to test points by learning sparse logistic regressors in user and item-set feature spaces. The second line maximizes node purity by ranking the preferred items of each user as highly as possible within its partition. Concretely, this is achieved by maximizing the Propensity-scored Normalized Discounted Cumulative Gain (PSnDCG) metric. PSnDCG is unbiased to missing items and boosts accuracy over rare and informative tail labels which are also frequently missing owing to their unfamiliarity to the users. $\mathcal{L}_{\text{rank}}(\mathbf{r}^-, \mathbf{y}_i^r)$ is the loss form of PSnDCG which is minimized by (1). The tight coupling between δ_i and the two regressors as well as the ranking terms help to learn node partitions that are both pure and accurate. Upon joint optimization, the two separators bring together those users having both similar user descriptions as well as similar revealed item preferences.

The time complexity for training SwiftXML is $O(N(T\log(N) + \hat{L})(\hat{D} + \hat{D}'))$, where N, T are the number of training instances and trained trees respectively; and $\hat{L}, \hat{D}, \hat{D}'$ are the average number of revealed items, non-zero user features and non-zero item-set features of a user, respectively. The above complexity includes both tree training times as well as tail label classifier training times, and is dominated by the former. Due to small values of $\hat{L}, T$ and a tractable $\log(N) * N$ dependence on the number of users, SwiftXML training is sufficiently fast and scalable.

Optimization: The discrete objective in (1) cannot be straight forwardly optimized with usual gradient descent techniques. Therefore, an efficient, iterative, alternating minimization algorithm is adopted which alternately optimizes over one of the four classes of variables $(\boldsymbol{\delta}, \mathbf{r}^\pm, \mathbf{w}_x, \mathbf{w}_z)$ at a time with the others held constant. Optimization over $\boldsymbol{\delta}$ with other variables held constant reduces (1) to N separate problems over individual δ_i variables which have simple closed form solutions:

$$\delta_i = \text{Sign}\Big(C_x \mathbf{w}_x^\top \mathbf{x}_i + C_z \mathbf{w}_z^\top \mathbf{z}_i + C_r(\mathcal{L}_{\text{rank}}(\mathbf{r}^+, \mathbf{y}_i^r)) - \mathcal{L}_{\text{rank}}(\mathbf{r}^-, \mathbf{y}_i^r))\Big)$$

Optimization over $\mathbf{r}^\pm$ with $\boldsymbol{\delta}, \mathbf{w}_x, \mathbf{w}_z$ fixed also has a closed form solution $\mathbf{r}^\pm = \text{rank}\Big(\sum_{i:\delta_i=\pm} I_L(\mathbf{y}_i^r)\mathbf{y}_i^r\Big)$, where rank($\mathbf{v}$) returns the indices of $\mathbf{v}$ in their descending order, and $I_L(\mathbf{y}_i^r)$ is a user-specific constant. Optimization over $\mathbf{w}_x$ or $\mathbf{w}_z$ while fixing the remaining variables reduces to standard L1 regularized logistic regression problems which can be efficiently solved using Liblinear [11]. In practice, the algorithm alternates between $\mathbf{r}^\pm$,$\mathbf{w}_x$ and $\mathbf{w}_z$ variables, interleaved with efficient $\boldsymbol{\delta}$ optimizations. Early stopping with just one iteration over all variables was found to be sufficient in practice.

Optimization time is empirically dominated by learning two L1 regularized logistic regressions, which have a combined cost of $O(N_{\text{node}} t(\hat{D} + \hat{D}'))$, where N_{node} is the number of training users in the node and t (usually set to 1) is the number of iterations.

Prediction: SwiftXML tree prediction involves routing the test user down the tree, starting from the root node until it reaches a leaf. At each visited internal node, the user is sent to left (right) child if the combined scores of the two separating hyperplanes, i.e. $\mathbf{C}_x \mathbf{w}_x^\top \mathbf{x} + C_z \mathbf{w}_z^\top \mathbf{z}$, is greater (lesser) than 0, with ties being broken randomly. When a leaf node is reached, the leaf's item distribution is used as the item preference scores. Thereupon, the individual tree scores are averaged to obtain the ensemble scores, $E_j(\mathbf{x}) = \sum_{t=1}^{T} P_j^t(\mathbf{x}, \mathbf{z})/T$ for the j item, where $P_j^t(\mathbf{x}, \mathbf{z})$ is the probability assigned by tree t to item j for the test point. These ensemble scores are further reranked by combining with the tail label classifier scores B_j as $s_j = \alpha \log E_j(\mathbf{x}) + (1-\alpha)\log B_j(\mathbf{x}) \quad \forall j \in \{1, .., L\}$. Finally the top scoring items are recommended to the test user.

The SwiftXML trees are well-balanced with $O(\log(N))$ depth. Consequently, tree prediction is efficient and its cost for a test user is $O(\log(N) * (\hat{D} + \hat{D}'))$. The overall complexity of the SwiftXML prediction algorithm is $O((T\log(N) + c) * (\hat{D} + \hat{D}'))$, where c is the number of top-scoring items being reranked by the base-classifiers.

Theoretical analysis of SwiftXML algorithm and optimization are beyond the scope of this work. Detailed derivations of the optimization steps and pseudocodes for SwiftXML training and prediction algorithms are presented in supplementary .

5 SPONSORED SEARCH ADVERTISING

Objective and motivation: Search engines generate most of their revenue via sponsored advertisements which are displayed alongside the organic search results. While creating an ad campaign, an advertiser usually provides minimalistic information such as a short description of the ad, url of the ad landing page, a few bid-phrases along with their corresponding bids. Bid-phrases are a set of search queries that are judged to be relevant to the ad, and hence bidded on by the advertisers. Since the potential monetizable queries are in the order of millions, the advertiser won't be able to provide exhaustive list of all relevant bid-phrases. To address this limitation, an ad retrieval system resorts to machine learning-based extended match techniques [9, 23] which suggest additional relevant search queries to bid on by leveraging advertiser provided information. This ad retrieval application is a natural fit for warm-start recommendations where the given bid phrases and historically known relevant search queries can be used to predict new search queries for the ad more accurately.

Ranking queries for ad retrieval: Ad retrieval task can be formulated as extreme multi-label learning problem by treating ad landing pages as data points and search queries as labels. Data point feature vectors are created by extracting bag-of-words representation from the ad landing pages. Relevant labels are generated from the historical click logs i.e. a search query is tagged to be relevant to an ad if the ad had received clicks when displayed for the query in the past. For the warm-start scenario, the advertiser provided bid-phrases as well as the queries that had led to clicks on the ad in the past are considered as revealed labels for the ad. Label features are extracted based on the CDSSM embeddings [25] of the bid-phrases or the search query phrases.

Label relevance weights: Not all ads which were clicked for a query in past are equally relevant or click-yielding to the query. One such example is the query "marco polo" which had received clicks mostly from hotel related ads along with a few clicks from book related ads. Treating all the clicked ads for this query as equally relevant results in the query "marco polo" being recommended for book related ads too, which in turn generates lower click through rates on the actual search engine traffic. To address this problem,

Table 1: Dataset statistics

Dataset	Train N	Features D	Labels L	Test M	Avg. labels per point	Avg. points per label
EURLex-4K	15,539	5,000	3,993	3,809	5.31	25.73
Wiki10-31K	14,149	101,938	30,935	6,613	17.25	11.58
AmazonCat-13K	1,186,239	203,882	13,330	306,782	5.05	566.01
CitationNetwork-36K	62,503	39,539	36,211	15,467	3.07	6.61
Amazon-79K	490,449	135,909	78,957	153,025	2.08	4.06
Wikipedia-500K	1,813,391	2,381,304	501,070	783,743	4.77	24.75

differential weights are assigned to each (query, ad) pair in the training data based on the ad's click generating potential for the query as measured by the normalized point-wise mutual information (nPMI). Let c_{ij} be number of historical clicks between ith ad and jth query. Normalized PMI is defined as $nPMI_{ij} = \frac{\log(p(i)p(j))}{\log p(i,j)} - 1$, where $p(i) = \sum_j c_{ij}/\sum_{i,j} c_{ij}$ and $p(i,j) = c_{ij}/\sum_{i,j} c_{ij}$

The nPMI values are higher for more relevant query-ad pairs in the training data and always lie between 0 and 1. SwiftXML is learned on the weighted training data, which helped to improve offline accuracy on the holdout data by 1% over the model trained on the unweighted data.

Inverted ad-index: Trained SwiftXML model was used to make recommendations for all the ads in the system. For every ad, the SwiftXML recommended queries having a score of above a threshold are inserted into an inverted ad index, to be retrieved when the corresponding query is entered in the search engine. A relatively high threshold of 0.7 was set on the SwiftXML scores to maximize precision. Additionally, the top two SwiftXML recommended queries were also inserted into the inverted ad index, since they were found to be mostly relevant despite scoring lower than the set threshold sometimes. This step improved recall, while maintaining the precision value, thus achieving 2% improvement in the offline F1-score on the holdout data, as compared to the global threshold-only approach.

Mapping onto bid-phrases: SwiftXML recommended queries need to be associated with one of the advertiser provided bid-phrases in order to choose appropriate bid value for the query, and also to allow the advertisers to track the campaign performance at the bid-phrase level. A query to bid-phrase click model is learned for this purpose, which assigns a click probability to a given (query, bid-phrase) pair. Training data of form (query, bid-phrase) is generated by considering historically clicked (query, ad) pairs and extracting the corresponding bid-phrase from the ad. A gradient boosted decision trees (GBDT) model is trained over the following features extracted for each (query, bid-phrase) pair: syntactic similarity features, historical click features (click counts at token level) and semantic similarity features (Word2vec [18] and CDSSM [25] embeddings). SwiftXML recommended query for an ad, is associated with the bid-phrase with maximum click probability, as predicted by the click model.

6 EXPERIMENTS

Experiments were carried out on benchmark datasets with up to half a million labels demonstrating that: (a) SwiftXML could be as much as 14% and 37% more accurate as compared to state-of-the-art extreme classification and warm-start recommendation algorithms, respectively; (b) SwiftXML could scale to extreme datasets beyond the scale of state-of-the-art warm-start recommendation algorithms that train on both document and label features and (c) deploying SwiftXML in a live system for sponsored search advertising on Bing led to significant gains in the click-through rate and quality of ad recommendations as well as simultaneous reductions in the bounce rate.

Datasets and features: Experiments were carried out on benchmark datasets containing up to 1.8 million training points, 2.3 million dimensional features and 0.5 million labels (see Table 1 for dataset statistics). The applications considered range from tagging Wikipedia articles (Wikipedia-500K), cataloging Amazon items into multiple Amazon product categories (AmazonCat-13K), item-to-item recommendation of Amazon products (Amazon-79K), paper

Table 2: SwiftXML can be up to 14% and 37% more accurate as compared to state-of-the-art extreme classifiers and warm-start recommendation algorithms respectively according to unbiased propensity-scored Precision@5 (PSP5). Results for PSP1, PSP3 and biased Precision@k are presented in the supplementary .

Dataset	Algorithm	Revealed Label Percentages			
		20%	40%	60%	80%
EURLex	WRMF	11.05	16.58	19.77	22.85
	SVD++	0.41	0.51	0.61	0.60
	BPR	1.13	1.01	0.86	2.24
	PfastreXML	48.21	48.64	51.13	52.46
	SLEEC	42.72	46.31	48.56	51.64
	PDSparse	43.79	45.72	46.02	49.87
	DiSMEC	47.03	48.30	50.54	51.63
	IMC	11.23	11.45	11.45	11.72
	Matchbox	0.50	–	1.00	1.09
	SwiftXML	**48.45**	**49.72**	**53.12**	**55.70**
Wiki10	WRMF	5.27	6.01	6.34	6.33
	PfastreXML	19.80	18.17	16.31	14.77
	SLEEC	12.42	12.57	12.28	12.14
	PDSparse	8.02	6.78	5.72	4.73
	DiSMEC	15.47	15.19	14.53	13.87
	IMC	2.36	3.42	3.87	3.98
	SwiftXML	**19.92**	**19.07**	**17.06**	**16.23**
AmazonCat	PfastreXML	76.32	75.80	75.17	76.30
	PDSparse	65.25	61.61	58.37	57.47
	SwiftXML	**77.17**	**81.10**	**83.77**	**87.83**
CitationNetwork	PfastreXML	15.39	15.19	15.30	15.24
	SLEEC	7.41	7.65	7.37	6.40
	PDSparse	14.65	14.48	14.05	14.21
	DiSMEC	**17.84**	17.78	18.06	18.49
	SwiftXML	16.92	**17.84**	**19.44**	**19.34**
Amazon	PfastreXML	36.39	36.14	36.61	35.40
	SLEEC	23.83	28.30	32.24	31.33
	PDSparse	34.12	33.57	33.54	32.85
	DiSMEC	**41.89**	41.94	42.54	41.86
	SwiftXML	37.69	**42.80**	**51.33**	**49.44**
Wikipedia	PfastreXML	33.34	33.35	33.22	35.22
	SwiftXML	**34.76**	**35.31**	**35.68**	**38.07**

Table 3: SwiftXML can be up to 14% and 4% more accurate according to unbiased propensity scored Precision@5 as compared to baseline extensions of PfastreXML incorporating label features via early and late fusion respectively. Results for other metrics, including biased Precision@k, are reported in the supplementary .

Dataset	Algorithm	Revealed Label Percentages 20%	40%	60%	80%
EURLex	PfastreXML-early	47.29	**49.82**	52.04	54.40
	PfastreXML-late	48.21	49.55	52.25	46.05
	SwiftXML	**48.45**	49.72	**53.12**	**55.70**
Wiki10	PfastreXML-early	19.59	18.17	16.31	14.94
	PfastreXML-late	19.80	18.03	16.31	14.65
	SwiftXML	**19.92**	**19.07**	**17.06**	**16.23**
AmazonCat	PfastreXML-early	74.56	78.53	79.77	81.52
	PfastreXML-late	76.23	78.20	81.43	85.40
	SwiftXML	**77.17**	**81.10**	**83.77**	**87.83**
CitationNetwork	PfastreXML-early	15.04	15.08	15.43	15.54
	PfastreXML-late	15.73	17.12	19.11	**19.86**
	SwiftXML	**16.92**	**17.84**	**19.44**	19.34
Amazon	PfastreXML-early	36.45	36.18	36.64	35.43
	PfastreXML-late	36.71	40.42	46.71	45.98
	SwiftXML	**37.69**	**42.80**	**51.33**	**49.44**
Wikipedia	PfastreXML-early	34.23	34.44	34.43	36.47
	PfastreXML-late	33.78	34.26	34.88	37.66
	SwiftXML	**34.76**	**35.31**	**35.68**	**38.07**

Table 4: SwiftXML could increase the relative click-through-rate (CTR) and relative quality of ad recommendation (QOA) by 10% while simultaneously reducing the bounce rate (BR) by 30% on sponsored search on Bing.

Algorithm	Relative CTR (%)	Relative QOA (%)	Relative BR (%)
Bing	100	100	100
PfastreXML	102	103	76
SwiftXML	110	112	69

citation recommendation (CitationNetwork-39K) and document tagging (EURLex-4K, Wiki10-31K). All the datasets can be publically downloaded from The Extreme Classification Repository [8]. Bag-of-words TF-IDF features provided on the Repository were used as the document (or user) features for each dataset. 500-dimensional word2vec embeddings [18] were used to generate the label features as these led to better results as compared to other word embedding models including glove [21] and phrase2vec [3]. Algorithms were evaluated under various warm-start conditions as more and more of a user's item preferences were revealed. This was simulated by randomly sampling 20%, 40%, 60% and 80% of the test labels and revealing them during training while the remaining labels were used for evaluation purposes as ground-truth.

Baseline algorithms: SwiftXML was compared to four types of algorithms for warm-start recommendation. First, SwiftXML was compared to WRMF [14], SVD++ [16] and BPR [24] which are collaborative filtering algorithms based on factorizing the label (ratings) matrix alone and do not leverage user or item features. Second, SwiftXML was compared to state-of-the-art extreme classifiers based on trees (PfastreXML [15]), embeddings (SLEEC [7]) and linear methods that learn a separate classifier per label (PDSparse [32] and DiSMEC [6]). The extreme classifiers improve upon the collaborative filtering methods by training on the user features along with the label (ratings) matrix. Third, SwiftXML was also compared to state-of-the-art recommender systems such as Inductive Matrix Completion (IMC) [20] and Matchbox [26] which extend collaborative filtering and extreme classification methods by leveraging user features, label (item) features and the label (ratings) matrix during both training and prediction. Finally, SwiftXML was compared to two alternate ways of extending the state-of-the-art tree based PfastreXML extreme classifier [15] to handle label features. In particular, PfastreXML-early uses early fusion to train PfastreXML on concatenated label and document features with the relative weighting of the two feature types being determined through validation. In contrast, PfastreXML-late uses late fusion to learn separate PfastreXML classifiers in the document and label feature spaces and then combines the two scores during prediction with relative weighting being again determined through validation.

Results are reported for the Mrec [2] recommender system library implementation of WRMF, the Mahout [1] implementation of SVD++ and the Matchbox implementation available on the Microsoft Azure cloud computing platform. The implementation of all the other algorithms was provided by the authors. Unfortunately, some algorithms do not scale to large datasets and results have therefore been reported for only those datasets to which an implementation scales. The relative performance of all the methods can be compared on the small scale EURLex dataset.

Hyper-parameters: In addition to the hyper-parameters of PfastreXML, SwiftXML has two extra hyper-parameters C_z, λ_z which weight the loss incurred over the label features in the node partitioning objective and the base classifiers respectively . Unfortunately, the Wikipedia-500K dataset is too large for hyper-parameter tuning through validation and therefore all algorithms were run with default values, with the default SwiftXML values kept same as in PfastreXML along with $C_z = \lambda_z = 1$. On the other datasets, the hyper-parameters for all the algorithms were tuned using fine grained validation so as to maximize the prediction accuracy on the validation set.

Evaluation metrics: Performance evaluation was done using precision@k and nDCG@k (with $k = 1, 3$ and 5) which are widely used metrics for extreme classification. Performance was also evaluated using propensity scored precision@k and nDCG@k (PSPk and PSNk with $k = 1, 3$ and 5) which have recently been shown to be unbiased, and more suitable, metrics [15] for extreme classification, tagging, ranking, recommendation, *etc.* The propensity model and values available on The Extreme Classification Repository were used. Results for all metrics apart from PSP5 are reported in the supplementary due to space limitations.

Results - prediction accuracy: Tables 2 and 3 compare prediction accuracy of SwiftXML to the various baseline algorithms as the percentage of labels revealed during training is varied from 20% to 80%. As can be seen in Table 2, SwiftXML's predictions can be up to 14% and 37% more accurate as compared to state-of-the-art extreme classifiers and warm-start recommendation algorithms respectively. The largest gains over existing extreme classifiers were observed for item-to-item recommendation on Amazon. This was because many Amazon products had unhelpfully brief product descriptions, translating into poor user features for existing extreme classifiers. In fact, in some extreme cases, some Amazon products had no product description whatsoever apart from the product name. However, the very same products had a number of other products that had frequently been bought together with them which contained some useful information. This was leveraged by SwiftXML as item features to make significantly better recommendations. SwiftXML was also able to make better predictions by leveraging such features even when a sufficiently verbose product description was available (see figure 1 for qualitative examples).

Table 3 also illustrates that SwiftXML can be more accurate as compared to early and late fusion methods for incorporating label features into PfastreXML by as much as 14% and 4% respectively. Early fusion has several limitations such as its tendency to overfit and an inherent bias towards the dense label features over sparse document features [30]. PfastreXML-late learns independent classifiers over the document and label features and therefore has suboptimal performance as compared to SwiftXML's which learns node separators in both spaces jointly.

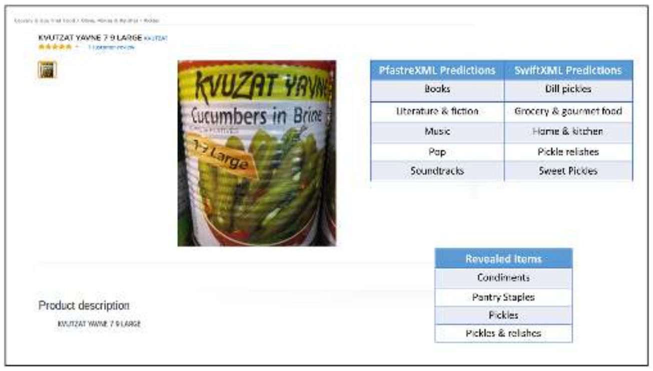
Results - scaling: As Tables 2 and 3 show, SwiftXML can efficiently scale to large datasets beyond the scale of warm-start recommendation algorithms such as IMC and Matchbox which also train on both document and label features. SwiftXML's training time is comparable to that of PfastreXML-early and PfastreXML-late, other techniques for handling warm-start problems based on both document and label features, but it's prediction time and model size can be lower. As compared to PfastreXML, SwiftXML's training time is 3-4x more in general but its prediction time and model size might be sometimes lower as it learns shorter better quality trees due to the extra information available. For instance, on the AmazonCat-13K dataset with 1.1 million training points and 13.3K labels, SwiftXML, PfastreXML-early, PfastreXML-late and PfastreXML,'s trained in 25, 19, 29 and 7 hours respectively on a single core of Intel Xeon 2.6 GHz server, while their model sizes were 10, 15, 19 and 16 GB respectively with prediction times being 4.2, 6.6, 4.7 and 4.3 milliseconds respectively. Note that SwiftXML training can be easily parallelized by growing each tree on a separate core, unlike algorithms such as Matchbox whose implementations run on only a single core. More parallelization can be attained in SwiftXML training by growing each node with the same tree depth on a separate core.

Sponsored Search Advertising: SwiftXML was used to rank the queries that might lead to a click on a given ad shown on Bing. While PfastreXML could only rank the queries on the basis of the text present on the ad-landing page, SwiftXML was able to leverage information about other queries that had already led to a click on the ad as well as queries that had been bid on by the advertiser for that page. Performance is measured in terms of click-through rate (CTR), bounce rate (QBR) and quality of ad recommendations (QOA). The bounce rate is defined as the percentage of times a user returns back immediately after viewing an ad landing page, indicating user dissatisfaction. The quality of ad recommendations measures the relevance of ad recommendations to a search query and is estimated by a query-ad relevance model trained on human labelled data. Table 4 compares SwiftXML to PfastreXML as well as a large ensemble of state-of-the-art methods for query ranking which are currently in production, referred to as Bing-ensemble. As can be seen, SwiftXML leads to upto 10% higher CTR and QOA, as well as upto 10% lower QBR as compared to both PfastreXML and Bing-ensemble. The training times of SwiftXML and PfastreXML algorithms were 140 and 200 hours respectively, while their model sizes were 3.7 and 3.8 GB respectively, with prediction times being 1.7 and 1.8 milliseconds per test point respectively.

Qualitative examples: Figure (1) illustrates the advantages of SwiftXML over PfastreXML through some representative examples. PfastreXML suffers from several limitations due to its sole reliance on user features, which are overcome by SwiftXML by leveraging revealed items and item feature information. First, the user features often fail to provide sufficient information for classification, with an extreme case being "(a) AmazonCat: Kvutzat Yavne pickles" whose web page contains limited text. In this case, PfastreXML recommends popular labels such as "Books" and "Literature & fiction" which are unfortunately irrelevant. On the other hand, SwiftXML leverages information about revealed tags such as "Pickles" and "Pickles & relishes" to recommend relevant tags such as "Dill pickles", "Grocery & gourmet food". Second, PfastreXML sometimes puts greater emphasis on the irrelevant, but frequent user features such as "language" and "Barron" in "(d) Amazon: Barron's IELTS ..." leading to irrelevant recommendations, such as "More useful french words" and "Spanish the Easy Way (Barron's E-Z)". In contrast, the information about revealed items such as "toefl" and "academic english" inform SwiftXML to emphasize more on the relevant user features such as "ielts" and "testing", resulting in useful IELTS preparation books being recommended to the customer. Third, PfastreXML is unable to disambiguate between homonyms such as "Fo" in the book title "(c) Amazon: Xsl Fo" versus "Fo" in the surname of dramatist "Dario Fo", both of which are among the user features. Hence, PfastreXML incorrectly recommends Dario Fo's plays instead of books on Xsl. SwiftXML resolves this ambiguity by using the context information from revealed books about "xsl" to recommend relevant books such as "Definitive XSL-FO" and "Learning XSLT". SwiftXML also leverages correlations between the revealed items and the relevant test items, to make accurate predictions. For example, the strong correlation between the revealed label "Marx" and novel label "Karl" in "(b) AmazonCat: Rosa Luxemburg..." is used by SwiftXML to correctly recommend "Karl". Furthermore, the revealed bid-phrases also help SwiftXML to accurately resolve advertiser intents such as selling archery items in "(e) Bing Ads: 3RiversArchery archery supplies" and selling pest bird control products in "(f) Bing Ads: Arcat pest control".

7 CONCLUSIONS

This paper extended the extreme classification formulation to handle warm-start scenarios by leveraging item features which can provide a rich, and complementary, source of information to the user features relied on by traditional extreme classifiers. The SwiftXML algorithm was developed for exploiting item features and label correlations as a simple, easy to implement and reproduce extension of the PfastreXML extreme classifier. Despite its simplicity,



KVUTZAT YAVNE 7 9 LARGE

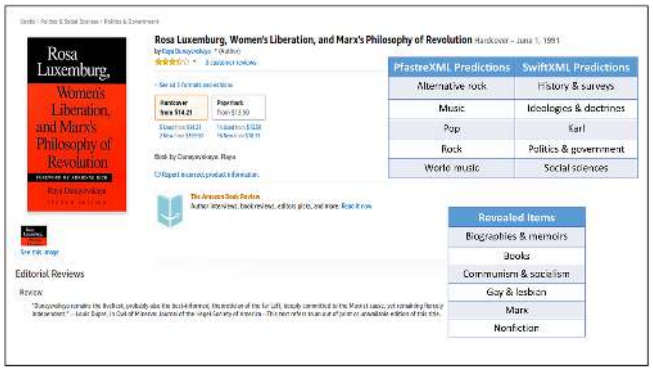
PfastreXML Predictions	SwiftXML Predictions
Books	Dill pickles
Literature & fiction	Grocery & gourmet food
Music	Home & kitchen
Pop	Pickle relishes
Soundtracks	Sweet Pickles

Revealed Items
Condiments
Pantry Staples
Pickles
Pickles & relishes

Product description

KVUTZAT YAVNE 7 9 LARGE

(a) AmazonCat: Kvutzat Yavne pickles



Rosa Luxemburg, Women's Liberation, and Marx's Philosophy of Revolution Hardcover – June 1, 1991

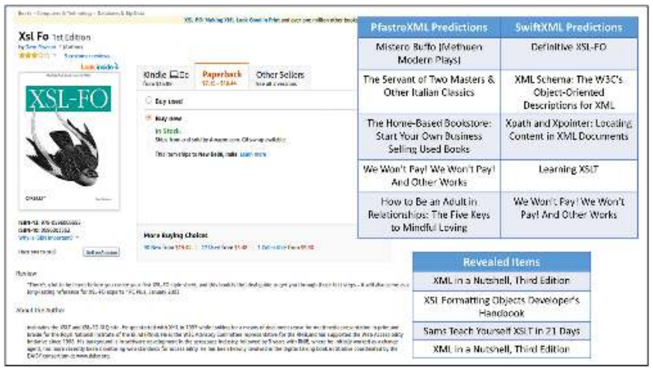
PfastreXML Predictions	SwiftXML Predictions
Alternative rock	History & surveys
Music	Ideologies & doctrines
Pop	Karl
Rock	Politics & government
World music	Social sciences

Revealed Items
Biographies & memoirs
Books
Communism & socialism
Gay & lesbian
Marx
Nonfiction

Editorial Reviews

Review

(b) AmazonCat: Rosa Luxemburg, Women's Liberation, and Marx's Philosophy of Revolution book

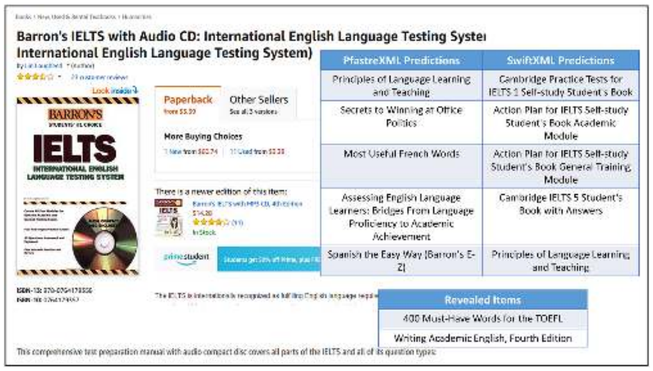


Xsl Fo 1st Edition

PfastreXML Predictions	SwiftXML Predictions
Mistero Buffo (Methuen Modern Plays)	Definitive XSL-FO
The Servant of Two Masters & Other Italian Classics	XML Schema: The W3C's Object-Oriented Descriptions for XML
The Home-Based Bookstore: Start Your Own Business Selling Used Books	Xpath and Xpointer: Locating Content in XML Documents
We Won't Pay! We Won't Pay! And Other Works	Learning XSLT
How to Be an Adult in Relationships: The Five Keys to Mindful Loving	We Won't Pay! We Won't Pay! And Other Works

Revealed Items
XML in a Nutshell, Third Edition
XSL Formatting Objects Developer's Handbook
Sams Teach Yourself XSLT in 21 Days
XML in a Nutshell, Third Edition

(c) Amazon: Xsl Fo book



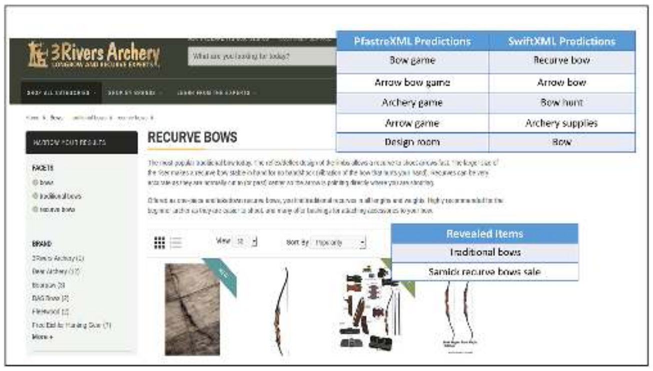
Barron's IELTS with Audio CD: International English Language Testing Syste International English Language Testing System)

PfastreXML Predictions	SwiftXML Predictions
Principles of Language Learning and Teaching	Cambridge Practice Tests for IELTS 1 Self-study Student's Book
Secrets to Winning at Office Politics	Action Plan for IELTS Self-study Student's Book Academic Module
Most Useful French Words	Action Plan for IELTS Self-study Student's Book General Training Module
Assessing English Language Learners: Bridges From Language Proficiency to Academic Achievement	Cambridge IELTS 5 Student's Book with Answers
Spanish the Easy Way (Barron's E-Z)	Principles of Language Learning and Teaching

Revealed Items
400 Must-Have Words for the TOEFL
Writing Academic English, Fourth Edition

This comprehensive test preparation manual with audio compact disc covers all parts of the IELTS and all of its question types.

(d) Amazon: Barron's IELTS with Audio CD book



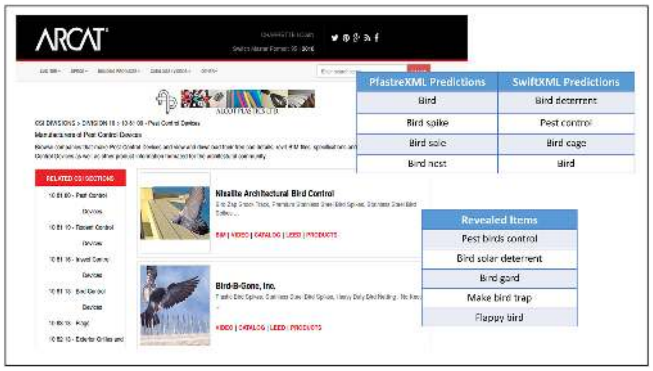
3Rivers Archery

RECURVE BOWS

PfastreXML Predictions	SwiftXML Predictions
Bow game	Recurve bow
Arrow bow game	Arrow bow
Archery game	Bow hunt
Arrow game	Archery supplies
Design room	Bow

Revealed Items
Traditional bows
Samick recurve bows sale

(e) Bing Ads: 3RiversArchery archery supplies



ARCAT

PfastreXML Predictions	SwiftXML Predictions
Bird	Bird deterrent
Bird spike	Pest control
Bird sole	Bird cage
Bird nest	Bird

Revealed Items
Pest birds control
Bird solar deterrent
Bird gard
Make bird trap
Flappy bird

Nixalite Architectural Bird Control

Bird-B-Gone, Inc.

(f) Bing Ads: Arcat pest control products

Figure 1: Item recommendations by PfastreXML and SwiftXML on AmazonCat, Amazon and Bing Ads: PfastreXML predictions are frequently irrelevant due to lack of informative user features (*e.g.* (a)), emphasis on the wrong features (*e.g.* (d)) and inability to disambiguate homonyms (*e.g.* (e)). SwiftXML leverages item correlations (*e.g.* "Marx" => "Karl" in (b)) and helpful information from revealed items and their features (*e.g.* (a)-(f)) to make much more accurate predictions. See text for more details. Figure best viewed under high magnification.

SwiftXML was shown to improve prediction accuracy on the Amazon item-to-item recommendation task by as much as 37% and 14% over state-of-the-art warm start recommendation algorithms and extreme classifiers respectively. Furthermore, live deployment for sponsored search advertising on Bing revealed that SwiftXML could increase the click-through rate and quality of ad recommendations by 10%, and reduce the bounce rate by 31% as compared to a large ensemble of state-of-the-art algorithms in production. The SwiftXML code will be made publically available.

REFERENCES

[1] [n. d.]. Apache Mahout. https://mahout.apache.org. ([n. d.]).

[2] [n. d.]. Mrec recommender systems library. http://mendeley.github.io/mrec. ([n. d.]).

[3] [n. d.]. Phrase2vec. https://github.com/zseymour/phrase2vec. ([n. d.]).

[4] D. Agarwal and B. C. Chen. 2009. Regression-based Latent Factor Models. In *KDD*. 19–28.

[5] R. Agrawal, A. Gupta, Y. Prabhu, and M. Varma. 2013. Multi-label Learning with Millions of Labels: Recommending Advertiser Bid Phrases for Web Pages. In *WWW*.

[6] R. Babbar and B. Schölkopf. 2017. DiSMEC: Distributed Sparse Machines for Extreme Multi-label Classification. In *WSDM*.

[7] K. Bhatia, H. Jain, P. Kar, M. Varma, and P. Jain. 2015. Sparse Local Embeddings for Extreme Multi-label Classification. In *NIPS*.

[8] K. Bhatia, H. Jain, Y. Prabhu, and M. Varma. [n. d.]. The Extreme Classification Repository: Multi-label Datasets & Code. ([n. d.]). http://manikvarma.org/downloads/XC/XMLRepository.html

[9] Y. Choi, M. Fontoura, E. Gabrilovich, V. Josifovski, M. Mediano, and B. Pang. 2010. Using landing pages for sponsored search ad selection. In *WWW*.

[10] M. Cissé, N. Usunier, T. Artières, and P. Gallinari. 2013. Robust Bloom Filters for Large MultiLabel Classification Tasks. In *NIPS*.

[11] R. E. Fan, K. W. Chang, C. J. Hsieh, X. R. Wang, and C. J. Lin. 2008. LIBLINEAR: A library for large linear classification. *JMLR* (2008).

[12] Z. Gantner, L. Drumond, C. Freudenthaler, and L. Schmidt-Thieme. 2012. Personalized Ranking for Non-Uniformly Sampled Items. In *Proceedings of KDD Cup 2011*. 231–247.

[13] D. Hsu, S. Kakade, J. Langford, and T. Zhang. 2009. Multi-Label Prediction via Compressed Sensing. In *NIPS*.

[14] Y. Hu, Y. Koren, and C. Volinsky. 2008. Collaborative Filtering for Implicit Feedback Datasets. In *ICDM*.

[15] H. Jain, Y. Prabhu, and M. Varma. 2016. Extreme Multi-label Loss Functions for Recommendation, Tagging, Ranking &#38; Other Missing Label Applications. In *KDD*.

[16] Y. Koren. 2008. Factorization Meets the Neighborhood: A Multifaceted Collaborative Filtering Model. In *KDD*.

[17] J. Lee, S. Bengio, S. Kim, G. Lebanon, and Y. Singer. 2014. Local collaborative ranking. In *WWW*.

[18] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. *CoRR* abs/1310.4546 (2013).

[19] P. Mineiro and N. Karampatziakis. 2015. Fast Label Embeddings for Extremely Large Output Spaces. In *ECML*.

[20] N. Natarajan and I. Dhillon. 2014. Inductive Matrix Completion for Predicting Gene-Disease Associations. In *Bioinformatics*.

[21] J. Pennington, R. Socher, and C. D. Manning. 2014. Glove: Global Vectors for Word Representation. In *EMNLP*. 1532–1543.

[22] Y. Prabhu and M. Varma. 2014. FastXML: A fast, accurate and stable tree-classifier for extreme multi-label learning. In *KDD*.

[23] H. Raghavan and R. Iyer. 2008. Evaluating vector-space and probabilistic models for query to ad matching. In *SIGIR Workshop on Information Retrieval in Advertising*.

[24] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI*.

[25] Y. Shen, X. He, J. Gao, L. Deng, and G. Mesnil. 2014. Learning semantic representations using convolutional neural networks for web search. In *WWW*.

[26] D. H. Stern, R. Herbrich, and T. Graepel. 2009. Matchbox: large scale online bayesian recommendations. In *WWW*.

[27] Y. Tagami. 2017. AnnexML: Approximate Nearest Neighbor Search for Extreme Multi-label Classification. In *KDD*. 455–464.

[28] J. Weston, S. Bengio, and N. Usunier. 2011. Wsabie: Scaling Up To Large Vocabulary Image Annotation. In *IJCAI*.

[29] J. Weston, A. Makadia, and H. Yee. 2013. Label Partitioning For Sublinear Ranking. In *ICML*.

[30] C. Xu, D. Tao, and C. Xu. 2013. A Survey on Multi-view Learning. *CoRR* (2013).

[31] I. E. H. Yen, X. Huang, W. Dai, P. Ravikumar, I. Dhillon, and E. Xing. 2017. PPDsparse: A Parallel Primal-Dual Sparse Method for Extreme Classification. In *KDD*. 545–553.

[32] I. E. H. Yen, X. Huang, K. Zhong, P. Ravikumar, and I. S. Dhillon. 2016. PD-Sparse: A Primal and Dual Sparse Approach to Extreme Multiclass and Multilabel Classification. In *ICML*.

[33] F. Zhang, T. Gong, Victor E. Lee, G. Zhao, C. Rong, and G. Qu. 2016. Fast Algorithms to Evaluate Collaborative Filtering Recommender Systems. *Know.-Based Syst.* (2016), 96–103.

[34] Y. Zhang and J. G. Schneider. 2011. Multi-Label Output Codes using Canonical Correlation Analysis. In *AISTATS*.