Build Faster with Less: A Journey to Accelerate Sparse Model Building for Semantic Matching in Product Search

Jiong Zhang
zhangjiong724@gmail.com
Amazon
Palo Alto, CA, USA

Yau-Shian Wang
yaushiaw@amazon.com
Amazon
Palo Alto, CA, USA

Wei-Cheng Chang
weicheng.cmu@gmail.com
Amazon
Palo Alto, CA, USA

Wei Li
zephyria.li@gmail.com
Amazon
Palo Alto, CA, USA

Jyun-Yu Jiang
jyunyu.jiang@gmail.com
Amazon
Palo Alto, CA, USA

Cho-Jui Hsieh
chohsieh@cs.ucla.edu
UCLA
Los Angeles, CA, USA

Hsiang-Fu Yu
rofu.yu@gmail.com
Amazon
Palo Alto, CA, USA

ABSTRACT

The semantic matching problem in product search seeks to retrieve all semantically relevant products given a user query. Recent studies have shown that extreme multi-label classification (XMC) model enjoys both low inference latency and high recall in real-world scenarios. These XMC semantic matching models adopt TF-IDF vectorizers to extract query text features and use mainly sparse matrices for the model weights. However, limited availability of libraries for efficient parallel sparse modules may lead to tediously long model building time when the problem scales to hundreds of millions of labels. This incurs significant hardware cost and renders the semantic model stale even before it is deployed. In this paper, we investigate and accelerate the model building procedures in a tree-based XMC model. On a real-world semantic matching task with 100M labels, our enhancements achieve over 10 times acceleration (from 3.1 days to 6.7 hours) while reducing hardware cost by 25%.

CCS CONCEPTS

• **Computing methodologies** → *Learning linear models*; **Parallel algorithms**; *Distributed algorithms.*

KEYWORDS

extreme multi-label classification, product search

ACM Reference Format:
Jiong Zhang, Yau-Shian Wang, Wei-Cheng Chang, Wei Li, Jyun-Yu Jiang, Cho-Jui Hsieh, and Hsiang-Fu Yu. 2023. Build Faster with Less: A Journey to Accelerate Sparse Model Building for Semantic Matching in Product Search. In *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management (CIKM '23), October 21–25, 2023, Birmingham, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3583780.3614661

1 INTRODUCTION

Matching in product search seeks to retrieve the most relevant products given an user issued query from the catalog containing billions of items. Lexical matching [29, 30] is currently the most widely used retrieval architecture in industry. To efficiently retrieve relevant candidates from a large search space, lexical models, such as BM25 [29], use inverted index to get documents containing same tokens as the query. These methods rely purely on token matching, making them vulnerable to morphological variants or spelling errors. Therefore, many search systems have semantic matching models in addition to lexical matching. Semantic matching models utilize query and document semantic meaning and can learn from customers' historical behavior. Embedding-based neural models [23, 36] learn query and document embeddings and use their inner product as the indicator of relevance. During inference, approximate nearest neighbor search (ANNs) [12, 21] is performed to get top relevant documents for a given query. In practice, however, considering the inference latency and model building cost, shallow neural networks are often used, which sacrifices model performance. In addition to these architectures, a recent study [5] demonstrated the efficiency and effectiveness of extreme multi-label classification (XMC) models in semantic matching. To handle extremely large output space, tree-based XMCs [27, 39, 40] recursively partition the label space to construct hierarchical label trees, enabling fast inference with time complexity logarithm to the label size. At the industry scale, tree-based XMC methods have shown significantly higher recall and low online inference latency compared with the dense two-tower models [5]. However, just as bi-encoder models need to re-index to adapt to the updated retrieval space, XMC methods, such as XR-Linear [39], need to be constantly refreshed to reflect latest user behavior and newly added products in the catalog. While bi-encoder training and inference have been well optimized, accelerating model building for XMC is under-studied.

In this work, we develop methods to accelerate large scale tree-based XMC model building. We focus on three components: query and label vectorization, label tree construction and matching model learning. An overview of our accelerations is presented in figure 1. In particular, we make the following contributions:

- With our efficient implementation in multiple component in XMC model building, we are able to reduce the building time of 100-million label semantic matching problem from 3.1 days to 6.7 hours (10 times acceleration) with 25% reduction in hardware cost.
- We demonstrate that our implementations are more efficient than the widely used off-the-shelf utilities, such as scipy,

CIKM '23, October 21–25, 2023, Birmingham, United Kingdom

ACM ISBN 979-8-4007-0124-5/23/10.
https://doi.org/10.1145/3583780.3614661

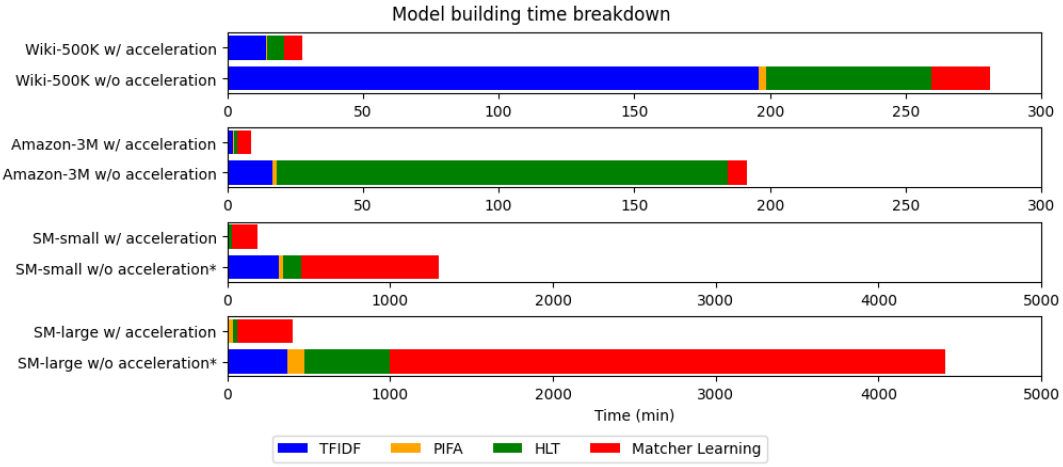

Figure 1: Comparison of model building time with and without our acceleration. From small benchmark data with 500K labels to the real world semantic matching task SM-large **with more than 100 million labels, we can achieve** 10× **to** 20× **acceleration over the baselines.**

sklearn, pytorch and IntelMKL. To support a wider range of data mining applications, we also provide standalone easy-to-use APIs and make all of our implementations open sourced in https://github.com/amzn/pecos.

2 RELATED WORKS

Bi-encoder models, or two-tower models, have been one of the most widely used architecture in semantic matching [22, 36]. These models leverage deep neural networks to encode query and products into a shared feature space and use maximum inner product search to retrieve relevant products for a given query. The bi-encoder models usually need to leverage strong encoders such as ResNet [13] or Transformers [7] to achieve good performance. These models are difficult to be deployed into large scale semantic matching systems because of significant cost to encode hundreds of millions of documents as well as large latency in query encoding. As an result, many real-world production systems, such as Youtube [9], Google Play [37] and e-commerce product search [23, 31], adopt shallow multi-layer perceptron (MLP) as two-tower encoders.

Extreme Multi-label Classification has received much attention in recent years. On one hand, recent studies have opened a gate to leverage powerful transformer models in XMC problems [8, 16]. Novel architectures are proposed to address the difficulty in fine-tuning transformers on very large label space. XR-Transformer [40] and CascadeXML [19] established the state of the art performance on public XMC benchmarks. In the industry community, partition based XMC models have successfully been adopted in applications such as dynamic search advertising [27, 28]. Recent studies also showed the efficiency of tree-based XMC method, such as XR-Linear [39], in handling semantic matching problem with search space up to 100 million in size. Tree-based semantic matching models [6] not only have shown significant performance gain over shallow bi-encoder methods, but also enjoys much lower inference latency.

In these models, sparse TF-IDF is used instead of DNNs to extract text features. This reduces the model size and improved inference throughput. However, while dense operations are highly optimized via BLAS [2] on CPUs and CUDA [24] on GPUs, existing implementations for most sparse operations are either missing or not efficient. For example, the general matrix matrix multiplication in most existing libraries only supports parallel computing while at least one of the matrices is dense. Intel MKL [35] supports parallel sparse-sparse matrix multiplication but it's not open-sourced. Another example is the hierarchical k-means with sparse features. As a half-century old algorithm, many previous works have studied various methods to accelerate the general K-means algorithm [1, 26, 34]. However, few works optimizes K-means while the features are highly sparse. Also, most libraries that have implementations of hierarchical k-means either does not support sparse features, such as Faiss [17], or cannot scale to large scale data, such as Scikit Learn [4].

3 PRELIMINARY: PECOS XR-LINEAR

Notations. Given an input text query $q \in \mathcal{Q}$, we want to find the most relevant k labels from a given label universe $\mathcal{Y} = \{\ell_1, ..., \ell_j, ..., \ell_L\}$ with size L. The training dataset $\{(q_i, \mathbf{y}_i) | q_i \in \mathcal{Q}, \mathbf{y}_i \in \{0,1\}^L, i \in [N]\}$ consists of relevant and irrelevant labels for N training instances.

In practice the number of labels L can scale to $10^7 - 10^8$. To efficiently learn and infer on such large label space, the tree-based XMC methods perform label space partitioning to divide and conquer the task. In these methods, the label space is partitioned and represented by an Hierarchical Label Tree (HLT) where each label is represented by a leaf node. The matching models (matchers) are then learned to navigate through the tree to find the most relevant leaf nodes. There are many works on tree-based XMC methods [14, 18, 27, 38, 40], yet not many have verified efficacy on industry scaled problems. In the scope of this paper, we consider a representative method PECOS XR-Linear [39], which has shown in previous studies the efficacy on industry level semantic matching problems [5]. The overall model building can be separated into 3 phases: query and label vectorization, HLT construction and matcher learning. We describe each of these components below.

Query and Label Vectorization: This step constructs the numerical representation for every input queries q_i and every label $\ell_j \in \mathcal{Y}$. In practice, the TF-IDF is used as query vectorizer. XR-Linear adopts the positive instance feaature aggregation (PIFA) to construct label features. In particular, given instance feature matrix $\mathbf{X} = [\mathbf{x}_1|...|\mathbf{x}_N]^\top \in \mathbb{R}^{N\times d}$ and label matrix $\mathbf{Y} = [\mathbf{y}_1|...|\mathbf{y}_N]^\top \in \{0,1\}^{N\times L}$, each label is represented by aggregating feature vectors from positive instances:

$$\mathbf{z}_\ell^{PIFA} = \frac{\mathbf{v}_\ell}{\|\mathbf{v}_\ell\|}, \text{where } \mathbf{v} = \sum_{j=1}^{L} \mathbf{Y}_{j\ell}\mathbf{x}_j = (\mathbf{X}^\top\mathbf{Y})_\ell, \forall \ell \in [L]. \quad (1)$$

Hierarchical Label Tree (HLT) Construction: At this phase, an HLT is constructed on the label space. The label indexing is usually done via hierarchical k-means clustering algorithms using the label features constructed in the previous step. Therefore, at level t of the T level HLT, the tree nodes consists of a new label space $\mathcal{Y}^{(t)} = \{\ell_1^{(t)}, ..., \ell_j^{(t)}, ..., \ell_{K_t}^{(t)}\}, t \in [T]$, where $K_T = L$.

Matcher Learning: At this phase, we learn the models to help us navigate through the HLT. At each level $t \in [T]$ of the hierarchical label tree, an one-versus-all (OVA) classifier $f^{(t)}(q, y)$, namely the matcher, is trained to score the relevance of any cluster $\ell_j^{(t)}$ given input q. In particular, the relevance score is computed via $f^{(t)}(q, y) = \sigma(\phi(q)^\top \mathbf{w}_j^{(t)})$, where $\mathbf{w}^{(t)} \in \mathbb{R}^{d\times K_t}$ is the model weight. The total learnable model weights is denoted as $\mathbf{W} = \{\mathbf{w}^{(t)}\}_{t\in[T]}$.

In fig 1, we present the time breakdown of the 4 important components with or without our acceleration. In the next sections, we will discuss each component individually and present our solution to accelerate each of them.

4 FAST TF-IDF VECTORIZATION

As a commonly used text feature extractor, there are off-the-shelf modules available in several open sourced packages, e.x. Scikit Learn. However, these implementations cannot be directly used in production systems because of the efficiency issue. Firstly, industry level product search systems have hundreds of millions of queries and it could cost several hours to construct TF-IDF feature for training queries. Secondly, online product search often happens under order of milliseconds and it's unacceptable for query vectorization alone to take more than that. Therefore, we implemented efficient TF-IDF vectorizer to satisfy these requirements.

Given a training corpus, the TF-IDF building procedure requires going over the corpus twice: first time to build vocabulary and then count the term and document frequencies.

Our C++ implementation based implementation adopts document-wise parallel computing for both steps. The implementation also allows hybrid tokenization such as combining character tri-gram with word bi-gram, where each tokenizer and TF-IDF vectorizer is learned independently. For large datasets, loading the corpus into memory would introduce large time and memory overhead. To handle such cases, we enabled out of memory TF-IDF training, where corpus file is processed by paralleled streaming fashion. In section 8.1, we present the acceleration of our TFIDF implementation over popular open sourced implementations, where our method reduces vectorization time by over 40 times on large datasets.

Algorithm 1 SPA-SpGEMM

Input: $\mathbf{A} \in \mathbb{R}^{M\times K}$ and $\mathbf{B} \in \mathbb{R}^{K\times N}$
Output: $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{M\times N}$
$\mathbf{C}.cptr[1] \leftarrow 1$
for $j = 1, ..., N$ **do**
  Initialize SPA: $SPA.\mathbf{w} = 0$, $SPA.\mathbf{b} = 0$, $SPA.nz = \{\}$
  for $k \in \{t \in [M] | \mathbf{B}_{t,j} \neq 0\}$ **do**
    for $i \in \{t \in [M] | \mathbf{A}_{t,k} \neq 0\}$ **do**
      $SPA.\mathbf{w}[i] += \mathbf{A}_{i,k}\mathbf{B}_{k,j}$
      $SPA.\mathbf{b}[i] = 1$
      $SPA.nz.insert(i)$
    end for
  end for
  for $i \in 1, ..., len(SPA.nz)$ **do**
    $\mathbf{C}.ridx[\mathbf{C}.cptr(j) + i] = SPA.nz[i]$
    $\mathbf{C}.data[\mathbf{C}.cptr(j) + i] = SPA.\mathbf{w}[SPA.nz[i]]$
  end for
  $\mathbf{C}.cptr(j + 1) \leftarrow \mathbf{C}.cptr(j) + len(SPA.nz)$
end for

5 ACCELERATING SPARSE GENERAL MATRIX MATRIX MULTIPLICATIONS

General Matrix Matrix Multiplications (GEMM) is one of the most fundamental linear algebra components used in almost all machine learning algorithms. Given matrix $\mathbf{A} \in \mathbb{R}^{M\times K}$ and $\mathbf{B} \in \mathbb{R}^{K\times N}$, it seeks to calculate the matrix multiplication result of $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{M\times N}$. Depending on the sparsity of the matrix $\mathbf{A}$ and $\mathbf{B}$, the GEMM can be categorized into dense-to-dense (GEMM), sparse-to-dense (SpDMM) and sparse-to-sparse (SpGEMM). Both GEMM and SpDMM will have dense result $\mathbf{C}$ and are now widely supported with highly-optimized implementations on CPUs/GPUs. Compared with others, SpGEMM is hard to optimize as the resulting pattern is irregular. As described in (1), the label feature construction via PIFA involves multiplying label matrix $\mathbf{Y}$ to the TFIDF feature matrix $\mathbf{X}$, with two matrices being highly sparse. In this section, we present an efficient implementation of SpGEMM algorithms that support parallelization on multi-core shared-memory machines.

There are two ways to perform SpGEMM leveraging the sparsity, namely InnerProduct-SpGEMM and ColumnWise-SpGEMM. The InnerProduct-SpGEMM performs sparse inner product for every entree of $\mathbf{C}$ and has time complexity $\mathbb{O}\left(MN \cdot \min\left(\frac{nnz(\mathbf{A})}{M}, \frac{nnz(\mathbf{B})}{N}\right)\right)$, where $\min\left(\frac{nnz(\mathbf{A})}{M}, \frac{nnz(\mathbf{B})}{N}\right)$ is the complexity of $< \mathbf{A}_{i,:}, \mathbf{B}_{:,j} >$ under sorted indices.

ColumnWise-SpGEMM compute $\mathbf{C}_{:,j}$ by aggregating columns of $\mathbf{A}$ corresponding to non zero entrees of $\mathbf{B}_{:,j}$. This gives time complexity $\mathbb{O}\left(\frac{nnz(\mathbf{A})nnz(\mathbf{B})}{K}\right)$. The sparsity condition ensures column-wise computation has much lower time complexity of the two.

Sparse Accumulator. The crucial part of columnwise SpGEMM is accumulating weighted columns of $\mathbf{A}$ onto sparse vector $\mathbf{C}_{:,j}$, a.k.a. the Sparse Accumulator (SPA) [11]. This is not trivial as the sparsity patterns of columns of $\mathbf{A}$ are irregular. To support constant time insertion and $\mathbb{O}(nnz)$ time gather, we implement abstract data type SPA that consists of value vector $\mathbf{w} \in \mathbb{R}^N$, nonzero indicator

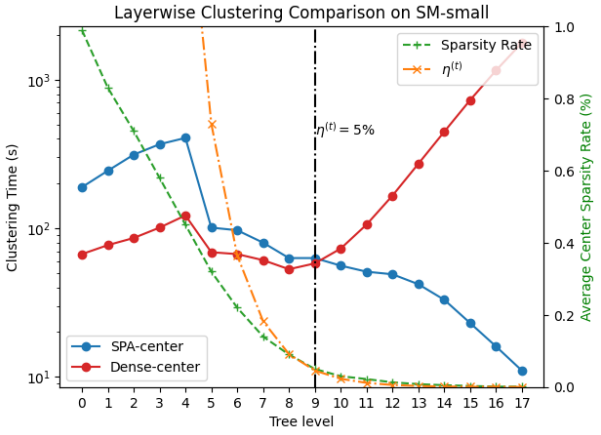

Figure 2: Layerwise clustering time along the HLT with dense and sparse cluster centers. As the number of clusters grow, cluster center becomes sparse and dense center initialization becomes the dominate cost.

vector $\mathbf{b} \in \{0,1\}^N$ and an unordered list nz storing non-zero indices. Finally, we present the entire SpGEMM algorithm implemented by the CSC matrices and the SPA. See details in Algorithm 1.

Parallel SpGEMM. Extending algorithm 1 into parallel setting is not trivial. Naively doing so by columnwise split $\mathbf{B}$ matrix can lead to double memory consumption of output matrix $\mathbf{C}$. This is because we need to copy each worker's local copy of submatrix $\mathbf{C}$ to the final aggregated results. To avoid this, we introduce a novel trick to estimate the upper bound of number of non-zeros in $\mathbf{C}$ and pre-allocate memory for $\mathbf{C}$. This resolves the double memory issue and save us the time for result copying.

In Section 8.1 we compare our implementation with state-of-the-art linear algebra libraries that implement the SpGEMM operation under both single threaded and multi-thread setting. Our proposed PECOS-SpGEMM achieves highest speedup.

6 LABEL SEMANTIC INDEXING

Upon constructing label representation $\mathbf{Z} \in \mathbb{R}^{d \times L}$, a hierarchical label tree (HLT) of depth T is constructed such that semantically similar labels are placed closer than irrelevant labels. The HLT is constructed via top-down B-array K-means clustering. At level $t \in \{1, 2, ..., T-1\}$, there are $K^{(t)} = B^t$ clusters, and at bottom level T the number of leaf nodes is $K^{(T)} = L$. The resulting HLT can be represented with a series of indexing matrices $\{\mathbf{C}^{(t)}\}_{t=1}^{T}$, where $\mathbf{C}^{(t)} \in \{0,1\}^{K_t \times K_{t-1}}$ is the adjacency matrix between two consecutive tree levels. At level t, the time complexity of one iteration of hierarchical B-means is given by

$$\mathbb{O}(dB^t) + \mathbb{O}(nnz(\mathbf{Z})) + \mathbb{O}(L \log \frac{L}{B^t}) \tag{2}$$

where the first term is the center initialization complexity, the second term is the cost for cluster center update and the third term is the cost of sorting node distances.

Sparse Accumulator for Cluster center update. One of the most time consuming parts in K-means algorithm is updating the cluster center, which involves accumulating the node embeddings within each individual cluster. For clustering with sparse features, this is tricky as the final sparsity pattern is not known until all cluster members have been accumulated. Most existing implementations avoid this by using a dense center even for sparse k-means. Although the accumulation can be done only on the non-zero dimensions of the node embeddings, the center initialization have complexity $\mathbb{O}(d)$ for each cluster and this overhead is not to be neglected when d and B^t is large. To address this issue, we leverage the sparse accumulator (SPA) described in section 5 to achieve fast cluster center initialization.

As illustrated in fig 2, using SPA centers result in significant acceleration when cluster centers are sparse but introduces overhead when the centers are close to dense, especially on top levels of the HLT. In practice we switch between SPA centers and dense centers using an estimation of the cluster center sparsity ratio by assuming no overlap amongst all nodes' features in all clusters:

$$\eta^{(t)} = \alpha \frac{nnz(\mathbf{Z})}{L} \frac{L}{B^t} \frac{1}{d} = \frac{nnz(\mathbf{Z})\alpha}{B^t d}. \tag{3}$$

When $\eta^{(t)} > \gamma$ we use dense accumulator for center update, and use SPA otherwise. Here γ and α are hyper-parameters that are often chosen as 5% and 10%. This will reduce the first term of (2) to $\mathbb{O}(\bar{p}B^t)$ where $\bar{p}$ is the average number of non-zeros in cluster centers.

Node sub-sampling Scheme. Researchers have studied accelerating large scale K-means clustering with bootstrapping on smaller sub-sampled data [3, 32]. In practice, we also observed that at top levels of the HLT, when there are millions of nodes in each of the cluster, clustering with an uniformly sampled subset of the labels results in similar clusters as using the full data. Therefore for large scale data, we adopt adaptive sampling scheme for different levels of the HLT clustering. In particular, we use a sampling rate $\beta^{(t)} = \frac{(t+b)}{T+b}$ that linearly increases by layer for clustering, where $b \in \mathbb{R}^+$ is the hyper-parameter to control the sampling rate.

With SPA centers and the node sub-sampling, at level t the time complexity for every k-means iteration is

$$\mathbb{O}(\bar{p}^{(t)}B^t) + \mathbb{O}(nnz(\mathbf{Z})\beta^{(t)}) + \mathbb{O}(\beta^{(t)}L \log \frac{\beta^{(t)}L}{B^t}). \tag{4}$$

On a semantic matching task with 100-million labels, we are able to reduce the clustering time from 8.7 hrs to 0.6 hrs (14.5X acceleration) with the same hardware. More empirical comparisons are presented in section 8.1.

7 DISTRIBUTED MATCHER LEARNING

To retrieve relevant labels from an extremely large space, XR-Linear learns matcher models to navigate through the tree. At level t of the HLT, a scoring function is learned $f^{(t)}(\mathbf{x}, \ell_j^{(t)}) = \sigma(< \mathbf{w}_{:,j}^{(t)}, \mathbf{x} >), \forall j \in K_t$, where $\mathbf{w}^{(t)} \in \mathbb{R}^{d \times K_t}$ is the model weight at level t. The positive nodes at level t is defined as $\mathbf{Y}^{(t)} = \text{binarize}(\mathbf{Y}^{(t+1)}\mathbf{C}^{(t+1)})$ where $\mathbf{Y}^{(T)} = \mathbf{Y}$ is the original label matrix. The query-label pairs considered in training $f^{(t)}$ is given by matching matrix $\mathbf{M}^{(t)} = \text{binarize}(\mathbf{Y}^{(t-1)}\mathbf{C}^{(t)\top})$ and the objective function at level t can be written as:

$$\min_{\mathbf{w}^{(t)}} \sum_{i=1}^{N} \sum_{\ell:\mathbf{M}^{(t)}_{i,\ell}\neq 0} \mathcal{L}(Y^{(t)}_{i,\ell}, \sigma(< \mathbf{w}^{(t)}_{:,\ell}, \mathbf{x} >)) + \lambda\|\mathbf{w}^{(t)}\|^2, \quad (5)$$

where $\mathcal{L}$ is a point-wise loss such as hinge loss, squared hinge loss or BCE loss.

Model Separability. Because $\mathbf{Y}^{(t)}$ and $\mathbf{M}^{(t)}$ are determined for a given HLT, the objective (5) independently columnwise

$$\min_{\mathbf{w}^{(t)}_{:,\ell}} \sum_{i:\mathbf{M}^{(t)}_{i,\ell}\neq 0} \mathcal{L}(Y^{(t)}_{i,\ell}, \sigma(< \mathbf{w}^{(t)}_{:,\ell}, \mathbf{x} >)) + \lambda\|\mathbf{w}^{(t)}_{:,\ell}\|^2. \quad (6)$$

This model separability feature can be used in single box parallel solving [10], where each column of $\mathbf{w}^{(t)}_{:,\ell}$ is optimized independently. We further exploit this feature and designed distributed solving scheme by leveraging the structure of the hierarchical label tree: Since all the non-zero entrees of $\mathbf{M}^{(t)}$ are within the same cluster, we can separate the hierarchical XMC problem into independent sub-probolems via tree structure splitting.

Meta and Sub-tree Training. The overall design of distributed XR-Linear follows the divide-and-conquer scheme: A pre-constructed HLT is separated into a meta tree (level 1 through $\hat{t}$) and $K = B^{\hat{t}}$ sub-trees. On each of the meta or sub-trees, an XR-Linear model can be trained independently. These $K+1$ XR-Linear models are then assembled to reconstruct the XR-Linear solution for the original XMC problem. Using the similar idea, the construction of HLT can also be separated into $K+1$ tasks and achieve distributed computing.

Load Balancing. Most XMC problems have "long tailed" label distribution [8], which leads to large variance in each sub-problem's training load. To address this issue, in practice, we choose the number of sub-problems K to be larger than the number of workers and perform load balancing to balance the load of each worker. In particular, the workload of an sub-tree is estimated by the total positive and negative training samples and we adopt the Longest-processing-time-first (LPT) algorithm to greedily assign the heaviest job to the lightest worker until all jobs are assigned.

8 EMPIRICAL RESULTS

All experiments are conducted on **x1.16xlarge** instances. In section 8.1, we compare the component-wise time cost between our implementation and widely used open sourced implementations. In section 8.2, we present the end-to-end model building time before and after the acceleration of each component.

Datasets. We conduct experiments on the largest two public XMC benchmark datasets (Wiki-500K and Amazon-3M) and industry level semantic matching datasets. Following the procedures in [6, 15, 20], we collect the semantic matching datasets from one of the largest e-commerce product search engine. In particular, we construct 12 months of search logs as the training set and use trailing 1 month' search log as the evaluation set. The resulting XMC training data (SM-large) are with around 100 million labels. We construct SM-small as an intermediate sized benchmark between Amazon-3M and SM-large via uniform sampling. Detailed data statistic are presented in table 1.

8.1 Component Wise Comparing

Text Vectorization. We conduct experiments comparing the training and prediction time of TF-IDF vectorizer on all datasets in Table 1 with results recorded in Table 2. We compare our implementation with baseline method from Scikit Learn [4] (Sklearn). For Wiki-500K and Amazon-3M, we use word tokenizer and set the max number of features same as [38]. For SM-small and SM-large, we follow the same setting as [6] and use combination of word unigram, word bigram and charactor trigram to build the TF-IDF feature with total of 4.2 million dimensions. On large dataset SM-small and SM-large, our TFIDF vectorizer is 30 times faster in training and over 90 times faster in predicting.

SpGEMM. We compare the proposed SPA-SpGEMM with state-of-the-art linear algebra libraries that implement the SpGEMM operation, including SciPy [33], IntelMKL [35] and Pytorch [25]. For all datasets we compute the matrix product between the feature matrix $\mathbf{Y}^\top \in \mathbb{R}^{L\times N}$ and $\mathbf{X} \in \mathbb{R}^{N\times d}$, where both matrices are presented in CSR format, except for Pytorch, where the matrices are converted to COO format before multiplication, as its CSR implementation is not able to scale to SM-small and SM-large. Given that we already included its backend MKL into our comparison, we omit Pytorch CSR from our results. Run time with 1 to 32 threads are presented in fig 3. Under both single-thread and multi-thread settings, our SPA-SpGEMM implementation outperforms other linear algebra libraries.

Table 1: Data statistics. N: the number of queries. L: the number of labels. $\bar{L}$: the average number of positive labels per instance. $\bar{n}$: average number of instances per label. d_{tfidf}: the sparse feature dimension of $\Phi_{tfidf}(\cdot)$.

Dataset	N	L	$\bar{L}$	$\bar{n}$	d_{tfidf}
Wiki-500K	1,779,881	501,070	4.75	16.86	2,381,304
Amazon-3M	1,717,899	2,812,281	36.04	22.02	337,067
SM-small	~200M	~20M	~50	~5	4,200,000
SM-large	~250M	~100M	~50	~20	4,200,000

Table 2: TF-IDF vectorizer training and predict time reported in min. Green texts in parentheses highlight the acceleration of our method over the corresponding Sklearn baseline.

Methods		Wiki-500K	Amazon-3M	SM-small	SM-large
Sklearn	fit	114.0	10.3	153.3	184.3
	transform	82.0	5.6	163.3	187.5
	Total	196.0	16.8	316.6	371.8
ours	fit	13.2	1.9	5.0	6.3
	transform	1.1	0.2	1.8	2.0
	Total	14.3(13.7×)	2.0(8.4×)	6.8(46.6×)	8.4(44.3×)

Table 3: Hierarchical Clustering time comparison in minutes. We use the green texts to highlight the acceleration over the Sklearn baselines and the blue texts to highlights acceleration over ours implementation without SPA and sampling.

Methods	Wiki-500K	Amazon-3M	SM-small	SM-large
Sklearn	61.0	166.3	N/A	N/A
ours	16.9(3.6×)	5.9(28.2×)	112.3(1.0×)	522.1(1.0×)
ours+SPA	6.9(8.8×)	3.6(46.2×)	20.6(5.5×)	109.4(4.8×)
ours+sampling	4.6(13.2×)	1.8(92.4×)	84.6(1.3×)	352.7(1.5×)
ours+both	3.8(16.1×)	1.4(118.8×)	9.7(11.6×)	36.0(14.5×)

Table 4: Detailed results with component wise breakdown.

Dataset	Enhancements			#instances	Time (min)				Total (min)	Cost ($)
	Vectorizer	SpGEMM	Clustering		TFIDF	PIFA	HLT	Training		
Wiki-500K	Sklearn	Scipy	Sklearn	1	196.0	2.6	61.0	21.5	281.2	31.3
	ours	Scipy	Sklearn	1	14.3	2.6	61.0	21.5	99.5	11.1
	ours	SPA-SpGEMM	Sklearn	1	14.3	0.3	61.0	21.5	97.2	10.8
	ours	SPA-SpGEMM	ours	1	14.3	0.3	16.9	21.5	53.1	5.9
	ours	SPA-SpGEMM	ours	4	14.3	0.3	16.9	6.8	35.5	15.8
	ours	SPA-SpGEMM	ours+SPA	4	14.3	0.3	9.1	6.8	30.5	13.6
	ours	SPA-SpGEMM	ours+SPA+sampling	4	14.3	0.3	6.3	6.8	27.8	12.4
Amazon-3M	Sklearn	Scipy	Sklearn	1	16.9	1.3	166.3	7.0	191.5	21.3
	ours	Scipy	Sklearn	1	2.0	1.3	166.3	7.0	176.7	19.6
	ours	SPA-SpGEMM	Sklearn	1	2.0	0.4	166.3	7.0	175.7	19.5
	ours	SPA-SpGEMM	ours	1	2.0	0.4	5.9	7.0	15.3	1.7
	ours	SPA-SpGEMM	ours	4	2.0	0.4	3.5	5.2	11.6	5.0
	ours	SPA-SpGEMM	ours+SPA	4	2.0	0.4	2.0	5.2	9.5	4.2
	ours	SPA-SpGEMM	ours+SPA+sampling	4	2.0	0.4	1.3	5.2	8.8	4.0
SM-small	Sklearn	Scipy	ours	1	316.6	26.0	112.3	846.7	1301.6	144.7
	ours	Scipy	ours	1	6.8	26.0	112.3	846.7	991.8	110.2
	ours	SPA-SpGEMM	ours	1	6.8	3.7	112.3	846.7	969.5	107.8
	ours	SPA-SpGEMM	ours	8	6.8	3.7	64.1	157.5	234.9	208.9
	ours	SPA-SpGEMM	ours+SPA	8	6.8	3.7	31.0	157.5	199.1	177.0
	ours	SPA-SpGEMM	ours+SPA+sampling	8	6.8	3.7	19.3	157.5	190.1	169.0
SM-large	Sklearn	Scipy	ours	1	371.8	104.7	522.1	3412.0	4410.5	490.2
	ours	Scipy	ours	1	8.4	104.7	522.1	3412.0	4047.2	449.8
	ours	SPA-SpGEMM	ours	1	8.4	27.5	522.1	3412.0	3970.1	441.3
	ours	SPA-SpGEMM	ours	8	8.4	27.5	186.0	342.0	564.0	501.5
	ours	SPA-SpGEMM	ours+SPA	8	8.4	27.5	71.0	342.0	449.0	399.2
	ours	SPA-SpGEMM	ours+SPA+sampling	8	8.4	27.5	26.0	342.0	404.0	359.2

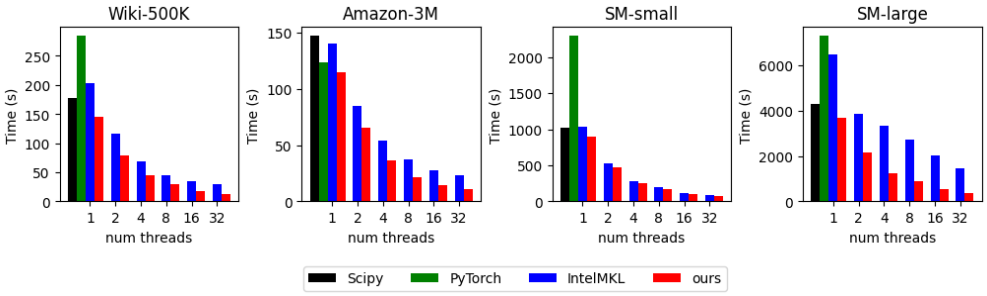

Figure 3: Comparison of SpGEMM runtime (averaged over 5 runs).

Hierarchical Clustering. Most available implementations for hierarchical clustering either does not support sparse features, e.g., FAISS [17], or rely on the condensed distance matrix which is not able to scale, e.g., Scipy [33]. Therefore, we compare our implementation of hierarchical k-means with the strong baseline method provided in Scikit Learn [4]. Both Sklearn and our methods leverage multi-CPU parallelism.

8.2 End2end acceleration

To understand the contribution of each enhancement, in table 4, we list the detailed breakdown of each of the steps in model building by adding our implemented enhancement one at a time. The money cost for model building is also computed by the hourly rate of x1.16xlarge instance of $6.67. Distributed training can give larger acceleration but only financially desirable for large datasets like SM-small and SM-large: with 8 x1.16xlarge instances, we are able to achieve 10 times acceleration with 26.7% reduction in hardware cost. In production, one needs to find a reasonable trade-off between time and hardware cost for the most efficient setting.

9 CONCLUSIONS

In this work, we present an end-to-end acceleration to the XMC model building under industry scale data. Our method achieves over 10 times acceleration while saving hardware cost by 25% on the industry level semantic matching tasks. Our enhanced modules, including TF-IDF vectorizer, SpGEMM, hierarchical k-means and distributed training setup/infrastructure are general enough for a larger spectral of data mining applications. We have made everything open sourced in https://github.com/amzn/pecos.

ACKNOWLEDGMENTS

We thank Amazon for supporting this work. We also thank Chih-Jen Lin for providing feedback on the manuscript.

REFERENCES

[1] David Arthur and Sergei Vassilvitskii. 2006. *k-means++: The advantages of careful seeding*. Technical Report. Stanford.

[2] L Susan Blackford, Antoine Petitet, Roldan Pozo, Karin Remington, R Clint Whaley, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, et al. 2002. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Software* 28, 2 (2002), 135–151.

[3] Paul S Bradley and Usama M Fayyad. 1998. Refining initial points for k-means clustering.. In *ICML*, Vol. 98. Citeseer, 91–99.

[4] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. 2013. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*. 108–122.

[5] Wei-Cheng Chang, Daniel Jiang, Hsiang-Fu Yu, Choon Hui Teo, Jiong Zhang, Kai Zhong, Kedarnath Kolluri, Qie Hu, Nikhil Shandilya, Vyacheslav Ievgrafov, et al. 2021. Extreme multi-label learning for semantic matching in product search. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*. 2643–2651.

[6] Wei-Cheng Chang, Daniel Jiang, Hsiang-Fu Yu, Choon-Hui Teo, Jiong Zhang, Kai Zhong, Kedarnath Kolluri, Qie Hu, Nikhil Shandilya, Vyacheslav Ievgrafov, Japinder Singh, and Inderjit S Dhillon. 2021. Extreme Multi-label Learning for Semantic Matching in Product Search. In *KDD*. ACM.

[7] Wei-Cheng Chang, Felix X. Yu, Yin-Wen Chang, Yiming Yang, and Sanjiv Kumar. 2020. Pre-training Tasks for Embedding-based Large-scale Retrieval. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkg-mA4FDr

[8] Wei-Cheng Chang, Hsiang-Fu Yu, Kai Zhong, Yiming Yang, and Inderjit S Dhillon. 2020. Taming pretrained transformers for extreme multi-label text classification. In *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 3163–3171.

[9] Paul Covington, Jay Adams, and Emre Sargin. 2016. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*. 191–198.

[10] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. 2008. LIBLINEAR: A library for large linear classification. *Journal of machine learning research* 9, Aug (2008), 1871–1874.

[11] John R Gilbert, Cleve Moler, and Robert Schreiber. 1992. Sparse matrices in MATLAB: Design and implementation. *SIAM journal on matrix analysis and applications* 13, 1 (1992), 333–356.

[12] Ruiqi Guo, Philip Sun, Erik Lindgren, Quan Geng, David Simcha, Felix Chern, and Sanjiv Kumar. 2020. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*. PMLR, 3887–3896.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.

[14] Kalina Jasinska-Kobus, Marek Wydmuch, Devanathan Thiruvenkatachari, and Krzysztof Dembczynski. 2021. Online probabilistic label trees. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 1801–1809.

[15] Jyun-Yu Jiang, Wei-Cheng Chang, Jiong Zhang, Cho-Jui Hsieh, and Hsiang-Fu Yu. 2022. Relevance under the Iceberg: Reasonable Prediction for Extreme Multi-label Classification. In *SIGIR '22: The 45th International ACM SIGIR Conference on Research and Development in Information Retrieval, Madrid, Spain, July 11 - 15, 2022*, Enrique Amigó, Pablo Castells, Julio Gonzalo, Ben Carterette, J. Shane Culpepper, and Gabriella Kazai (Eds.). ACM, 1870–1874. https://doi.org/10.1145/3477495.3531767

[16] Ting Jiang, Deqing Wang, Leilei Sun, Huayi Yang, Zhengyang Zhao, and Fuzhen Zhuang. 2021. LightXML: Transformer with Dynamic Negative Sampling for High-Performance Extreme Multi-label Text Classification. In *AAAI*.

[17] Jeff Johnson, Matthijs Douze, and Hervé Jégou. 2019. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data* 7, 3 (2019), 535–547.

[18] Sujay Khandagale, Han Xiao, and Rohit Babbar. 2019. BONSAI-Diverse and Shallow Trees for Extreme Multi-label Classification. *arXiv preprint arXiv:1904.08249* (2019).

[19] Siddhant Kharbanda, Atmadeep Banerjee, Erik Schultheis, and Rohit Babbar. 2022. CascadeXML: Rethinking Transformers for End-to-end Multi-resolution Training in Extreme Multi-label Classification. In *Conference on Neural Information Processing Systems*.

[20] Hanqing Lu, Youna Hu, Tong Zhao, Tony Wu, Yiwei Song, and Bing Yin. 2021. Graph-based Multilingual Product Retrieval in E-Commerce Search. In *NAACL-HLT (Industry Papers)*. 146–153. https://doi.org/10.18653/v1/2021.naacl-industry.19

[21] Y. A. Malkov and D. A. Yashunin. 2020. Efficient and Robust Approximate Nearest Neighbor Search Using Hierarchical Navigable Small World Graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42, 4 (2020), 824–836.

[22] Tri Nguyen, Mir Rosenberg, Xia Song, Jianfeng Gao, Saurabh Tiwary, Rangan Majumder, and Li Deng. 2016. MS MARCO: A Human-Generated MAchine Reading COmprehension Dataset. (2016).

[23] Priyanka Nigam, Yiwei Song, Vijai Mohan, Vihan Lakshman, Weitian Ding, Ankit Shingavi, Choon Hui Teo, Hao Gu, and Bing Yin. 2019. Semantic product search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2876–2885.

[24] NVIDIA, Péter Vingelmann, and Frank H.P. Fitzek. 2020. CUDA, release: 10.2.89. https://developer.nvidia.com/cuda-toolkit

[25] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[26] Dan Pelleg, Andrew W Moore, et al. 2000. X-means: Extending k-means with efficient estimation of the number of clusters.. In *Icml*, Vol. 1. 727–734.

[27] Yashoteja Prabhu, Anil Kag, Shrutendra Harsola, Rahul Agrawal, and Manik Varma. 2018. Parabel: Partitioned label trees for extreme classification with application to dynamic search advertising. In *Proceedings of the 2018 World Wide Web Conference*. 993–1002.

[28] Yashoteja Prabhu, Aditya Kusupati, Nilesh Gupta, and Manik Varma. 2020. Extreme regression for dynamic search advertising. In *Proceedings of the 13th International Conference on Web Search and Data Mining*. 456–464.

[29] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389.

[30] Stephen E Robertson and Steve Walker. 1994. Some simple effective approximations to the 2-poisson model for probabilistic weighted retrieval. In *SIGIR'94*. Springer, 232–241.

[31] Liuyihan Song, Pan Pan, Kang Zhao, Hao Yang, Yiming Chen, Yingya Zhang, Yinghui Xu, and Rong Jin. 2020. Large-scale training system for 100-million classification at alibaba. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2909–2930.

[32] Aurora Torrente and Juan Romo. 2021. Initializing k-means clustering by bootstrap and data depth. *Journal of Classification* 38, 2 (2021), 232–256.

[33] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. 2020. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods* 17 (2020), 261–272. https://doi.org/10.1038/s41592-019-0686-2

[34] Kiri Wagstaff, Claire Cardie, Seth Rogers, Stefan Schrödl, et al. 2001. Constrained k-means clustering with background knowledge. In *Icml*, Vol. 1. 577–584.

[35] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. 2014. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi™*. Springer, 167–188.

[36] Lee Xiong, Chenyan Xiong, Ye Li, Kwok-Fung Tang, Jialin Liu, Paul N Bennett, Junaid Ahmed, and Arnold Overwijk. 2020. Approximate Nearest Neighbor Negative Contrastive Learning for Dense Text Retrieval. In *International Conference on Learning Representations*.

[37] Ji Yang, Xinyang Yi, Derek Zhiyuan Cheng, Lichan Hong, Yang Li, Simon Xiaoming Wang, Taibai Xu, and Ed H Chi. 2020. Mixed negative sampling for learning two-tower neural networks in recommendations. In *Companion Proceedings of the Web Conference 2020*. 441–447.

[38] Ronghui You, Zihan Zhang, Ziye Wang, Suyang Dai, Hiroshi Mamitsuka, and Shanfeng Zhu. 2019. Attentionxml: Label tree-based attention-aware deep model for high-performance extreme multi-label text classification. *Advances in Neural Information Processing Systems* 32 (2019).

[39] Hsiang-Fu Yu, Kai Zhong, Jiong Zhang, Wei-Cheng Chang, and Inderjit S Dhillon. 2022. PECOS: Prediction for Enormous and Correlated Output Spaces. *Journal of Machine Learning Research* (2022).

[40] Jiong Zhang, Wei-Cheng Chang, Hsiang-Fu Yu, and Inderjit S Dhillon. 2021. Fast Multi-Resolution Transformer Fine-tuning for Extreme Multi-label Text Classification. *Advances in Neural Information Processing Systems* 34, 7267–7280.