Efficient Graph based Recommender System with Weighted Averaging of Messages

Faizan Ahemad
fahemad3@gmail.com
Amazon India
Bengaluru, Karnataka, India

ABSTRACT

We showcase a novel solution to a recommendation system problem where we face a perpetual soft item cold start issue. Our system aims to recommend demanded products to prospective sellers for listing in Amazon stores. These products always have only few interactions thereby giving rise to a perpetual soft item cold start situation. Modern collaborative filtering methods solve cold start using content attributes and exploit the existing implicit signals from warm start items. This approach fails in our use-case since our entire item set faces cold start issue always. Our Product Graph has over 500 Million nodes and over 5 Billion edges which makes training and inference using modern graph algorithms very compute intensive.

To overcome these challenges we propose a system which reduces the dataset size and employs an improved modelling technique to reduce storage and compute without loss in performance. Particularly, we reduce our graph size using a filtering technique and then exploit this reduced product graph using **W**eighted **A**veraging of **M**essages over **L**ayers (**WAML**) algorithm. **WAML** simplifies training on large graphs and improves over previous methods by reducing compute time to $\frac{1}{7}$ of LightGCN [8] and $\frac{1}{26}$ of Graph Attention Network (GAT) [20] and increasing recall@100 by **66%** over LightGCN and **2.3**x over GAT.

CCS CONCEPTS

• **Information systems** → **Recommender systems**; *Personalization*; • **Computing methodologies** → *Neural networks*; Feature selection; • **Applied computing** → Online shopping; • **Human-centered computing** → *Collaborative filtering*.

KEYWORDS

graph neural networks, recommendation system, compute efficiency, data efficiency, personalization

ACM Reference Format:
Faizan Ahemad. 2022. Efficient Graph based Recommender System with Weighted Averaging of Messages. In *The Second International Conference on AI-ML Systems (AIMLSystems 2022), October 12–15, 2022, Bangalore, India.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3564121.3564127

AIMLSystems 2022, October 12–15, 2022, Bangalore, India

ACM ISBN 978-1-4503-9847-3/22/10…$15.00
https://doi.org/10.1145/3564121.3564127

1 INTRODUCTION

Modern recommenders use collaborative filtering (CF) [27], which view user-product interaction data as a partially observed matrix and then try to predict the unseen observations using algorithms such as Matrix Factorization [11], NMF [13], DeepFM [6], Neural CF [9] and Graph Convolution Matrix Completion (GCMC) [18]. These methods perform well for warm start products which already have many interactions, but they fail to recommend new products with no interactions. Various neural network methods, such as DropoutNet [21] and CLCRec [24], are used to inject content-based user and product attributes into the CF algorithms to solve cold start issues.

Graph based CF methods [23] view matrix completion as a link prediction problem on graphs like GCMC [18], GraphSAGE [7], PinSAGE [26], Alibaba's recommender system [22], Graph Attention Network (GAT) [20] and LightGCN [8]. All graph methods [5] not only look at the current user and product but also their neighbouring users and products over multiple hops on the product graph. Further, the product graph can be constructed using a variety of nodes and edges apart from the user and product nodes. For example, social network data can be incorporated by linking users who are friends or followers in the social network [4]. User-product interactions can be predicted as link prediction task between user and product nodes.

In our usecase, we faced the challenge of **perpetual soft item cold start** where every product from candidate set of recommendations have few previous interactions, see Table 1, but not zero interactions, once the product obtains interactions above a small set threshold it is removed from candidate set of recommendations. CF systems optimize performance on warm start items and solve cold start by using content attributes and links with other warm start items, but we don't have any warm start items. Cold start methods only use content attributes and fail to exploit the graph structure in our product data. Neither of these methods perform well in our case where only few interactions are present for all candidate items. Various features required by our usecase are listed in Table 2 along with details on which recommendation models support these features.

We propose improvements over Collaborative Filtering (CF), cold start methods and Graph methods for our domain of **perpetual soft item cold start** and improve algorithmic performance as well as reduce compute and data storage requirements by making the below contributions:

(1) **W**eighted **A**veraging of **M**essages over **L**ayers (**WAML**), refer Section 2, obtains a middle ground between warm start and cold start on our perpetual soft item cold start system. WAML operates over our product graph from Section 2.1

Table 1: Count of products in our candidate dataset $\mathcal{P}_\mathrm{r}$ and how many interactions each product has.

Product Type	No. of Interactions	% of Products
Cold Start	0	14.2%
Soft Cold Start	≤3	47.9%
Mildly Warmed	3-10	37.8%
Fully Warmed	≥10	0.00%
All Products	~2.8	100.%

which captures relationships in product catalogue and user-product interactions (clicks, views etc.). For cold start, WAML ensures that product content embeddings aren't eclipsed by collaborative signals unlike other CF systems. We infuse product data into WAML embeddings through BERT [3] embeddings of product title and description.

(2) **Reduce product graph size** as described in Section 2.1 and Figure 1, using an innovative graph reduction mechanism from billions of edges and nodes to 250x reduction in edges and 66x reduction in nodes, providing 45x reduction in overall compute while retaining 79.5% of maximum recall, see last 2 rows of Table 4.

(3) WAML doesn't use complex neural networks for graph neighbourhood feature aggregation, see Figure 2a, since feature transformations and non-linear activation on each graph neural network layer has no positive effect on collaborative filtering [8]. It only performs **layer-wise weighted combination** of node and aggregated neighbourhood features.

(4) Inspired by Semi-supervised learning [2, 25] and contrastive learning for recommendations [24] we train WAML embeddings using using a **graph contrastive loss** over the link prediction task similar. This enables WAML to converge faster while producing K-Nearest Neighbour friendly user and product vectors.

2 METHOD

2.1 Building the Product Graph

We have a set of customers $\mathcal{C}$ ($\geq 200M$)[1] interacting with products $\mathcal{P}$ ($\geq 200M$) which belong to product categories $\mathcal{A}$, and sellers $\mathcal{S}$ ($\geq 300K$) selling these products[2]. Customer-product interactions[3] form edges $E^{\mathcal{C}\cup\mathcal{P}}$ ($\geq 5B$) which are the largest edge set in our data, seller-product offerings form edges $E^{\mathcal{S}\cup\mathcal{P}}$ ($\sim 200M$), and product to product category mappings form edges $E^{\mathcal{A}\cup\mathcal{P}}$ ($\sim 100M$). Let $\mathcal{P}_\mathrm{r} \subset \mathcal{P}$ be our candidate set of products from which we make recommendations which is $\sim 500K$. Since our recommendations are only for sellers, we removed all customer nodes $\mathcal{C}$ by linking any two products bought by same customer and only take those product to product links which occur over 500 times in our dataset. We remove any product node not in 1-hop neighbourhood of any node in $\mathcal{P}_\mathrm{r}$ (our candidate item set) and obtain a product set of candidate items $\mathcal{P}_\mathrm{r}$ with their 1-hop neighbours from $\mathcal{P}$ forming our training product set $\mathcal{P}_\mathrm{t} \mid \mathcal{P}_\mathrm{r} \subset \mathcal{P}_\mathrm{t} \subset \mathcal{P}$ and product to product edges $E^{\mathcal{P}_\mathrm{t}}$.

Our final graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is composed of nodes $\mathcal{V} = \mathcal{S} \cup \mathcal{P}_\mathrm{t} \cup \mathcal{A}$ and edges $\mathcal{E} = E^{\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{A}\cup\mathcal{P}_\mathrm{t}}$. Our graph reduction mechanism enables us to reduce our graph size from $5B$ edges and $400M$ nodes to just $20M$ edges (250x lower) and $6M$ nodes (66x lower). For an example of this process see Figure 1, we provide the detailed steps of this approach in Appendix A. Table 5 provides details on approximate relative count of edges and nodes we obtain before and after filtering.

2.2 WAML architecture

Each node $v \in \mathcal{V}$ in our graph is associated with real-valued attributes $x_v \in \mathbb{R}^d$ known as content features. For products $\mathcal{P}$ this is obtained by using BERT encoder on product title and description. For sellers $\mathcal{S}$ and product categories $\mathcal{A}$ nodes, we fill these with zeros. Our candidate products $\mathcal{P}_\mathrm{r}$ have very less interactions with our sellers $\mathcal{S}$, see Table 1. As such our algorithm must be able to balance usage of content features with sparse collaborative signals from the graph. We denote neighbour nodes of a node $v \in \mathcal{V}$ as $\mathcal{N}(v)$, each node's representation after a layer of graph convolution as h_v^i where $i \in \{1, \ldots, K\}$ and K is the number of graph convolution layers in the model, node's neighbourhood representations as $\{h_u^i : u \in \mathcal{N}(v)\}$. We list the differences between our method WAML and other graph based methods such as LightGCN and GAT in Table 3 in Appendix B.

Our WAML architecture is depicted in Figure 2. Node representations on starting are a combination of three different embeddings, non-trainable node identifier hashes, node type identifier hashes, content embedding from BERT. Non-trainable node identifier hash HASH(Node Id) $\in \mathbb{R}^d$ replaces trainable per node embeddings, which scale well with increasing nodes in graph. We also incorporate node type hashes HASH(Node type) $\in \mathbb{R}^d$ to ensure WAML can treat each node type independently and process seller, product and product category nodes separately if needed. We use content embedding x_v for all product nodes $\mathcal{P}_\mathrm{t}$ from BERT output of each product's title and description while setting content embedding to zeros for sellers $\mathcal{S}$ and product categories $\mathcal{A}$. These embeddings before being fed into WAML layers are denoted by $h_v^0 : v \in \mathcal{V}, h_v^0 \in \mathbb{R}^d$ where h_v^0 = HASH(Node Id)+HASH(Node type)+x_v. Next, WAML layers, see Figure 2a, convolve over each node $v \in \mathcal{V}$ with the first layer input as h_v^0 and layer i output as h_v^i. During each convolution a node v combines it's own features h_v^i with it's aggregated neighbour nodes features $h_{\mathcal{N}(v)}^i = \sum_u^{\mathcal{N}(v)} h_u^i$ using a weighted sum operation. Both node features h_v^i and aggregated neighbourhood features $h_{\mathcal{N}(v)}^i$ are L2 normalized before they are combined. Each WAML layer i has a parameter α^i which controls how much of neighbourhood features are integrated into the current node. For $\forall v \in \mathcal{V}$ the WAML layer i performs the following steps:

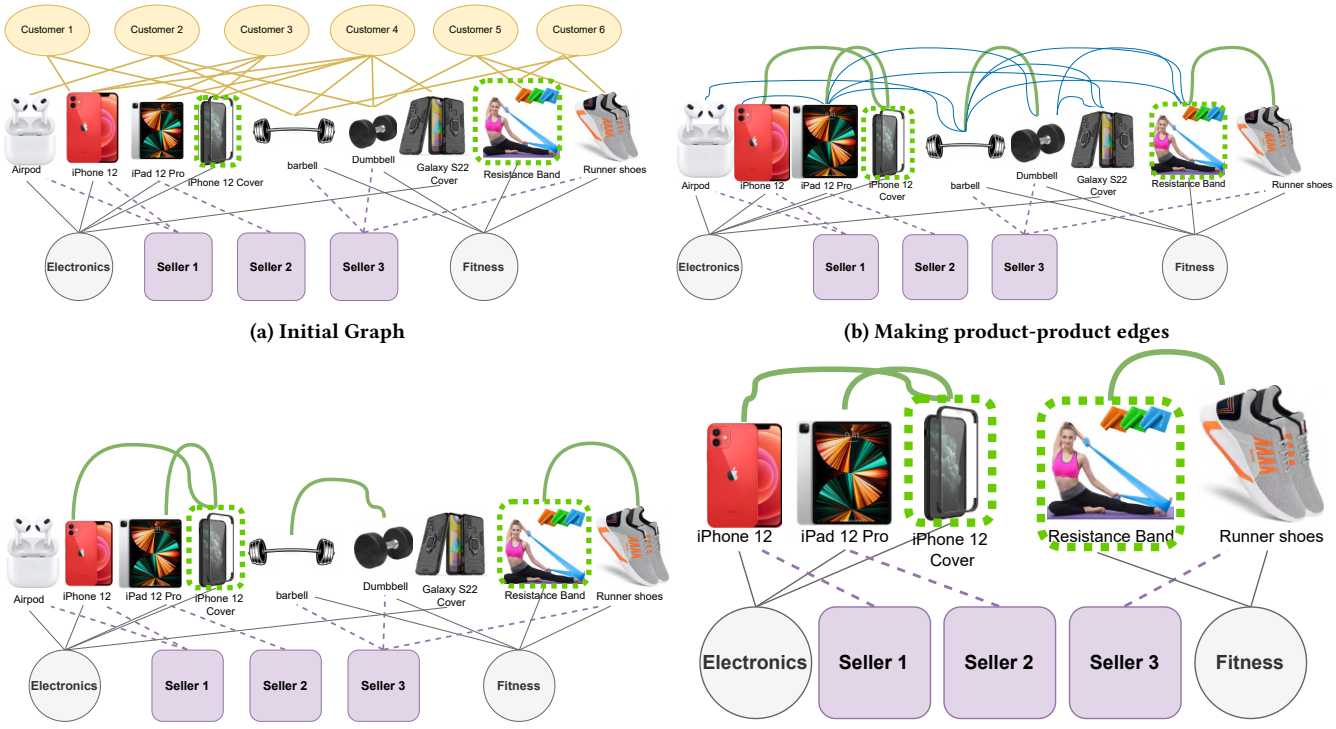
[1] M = million (10^6); K = thousand (10^3); B = Billion (10^9)

[2] Numbers given here are for demonstrative purposes to give a sense of how our method down-samples the product graph. These are not actual business derived numbers of any e-commerce store.

[3] We anonymise the customers for the product graph before using their interactions to build the graph, any customer data used in our system cannot be de-anonymised.

Table 2: Features required by our usecase and which methods support them.

Feature	NCF	MF (SVD)	GCMC or GAT	Content based	GAT + DropoutNet	LightGCN	WAML (ours)
Cold Start	✗	✗	✗	✓	✓	✗	✓
Soft Cold Start	✗	✗	✗	✗	✓	✗	✓
Use product attributes	✓	✗	✓	✓	✓	✗	✓
Exploit Graph Structure	✗	✗	✓	✗	✓	✓	✓
Heterogenous Graph	✗	✗	✓	✗	✓	✗	✓
Simple Implementation	✓	✓	✗	✓	✗	✓	✓
Trains Fast	✓	✓	✗	✓	✗	✓	✓
Fast Inference with KNN	✗	✓	✗	✓	✗	✓	✓
Scaling with Edge count	✓	✓	✗	✓	✗	✓	✓
Scaling with Node count	✓	✓	✗	✓	✗	✗	✓
Scaling with Node degree	✓	✓	✗	✓	✗	✓	✓
Low Memory usage	✓	✓	✗	✓	✗	✗	✓
Low Overall Compute	✗	✓	✗	✓	✗	✓	✓



(a) Initial Graph

(b) Making product-product edges

(c) Edge threshold filter

(d) Remove products not connected to candidate set $\mathcal{P}_\mathbf{r}$

Figure 1: Process to reduce edges and nodes in graph. Products surrounded in green dotted boxes belong to our candidate set $\mathcal{P}_\mathbf{r}$. (a) Initial Graph. (b) Remove customers and link products. (c) Remove product-product edges below a fixed threshold here 2. (d) Remove products not connected to candidate set $\mathcal{P}_\mathbf{r}$, forming our training product set $\mathcal{P}_\mathbf{t}|\mathcal{P}_\mathbf{r} \subset \mathcal{P}_\mathbf{t}$.

$$h_v^i = \frac{h_v^i}{\left\|h_v^i\right\|}$$

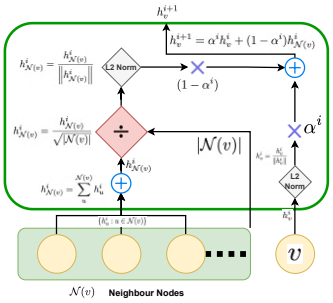
$$h_{\mathcal{N}(v)}^i = \sum_u^{\mathcal{N}(v)} h_u^i$$

$$h_{\mathcal{N}(v)}^i = \frac{h_{\mathcal{N}(v)}^i}{\sqrt{|\mathcal{N}(v)|}}$$

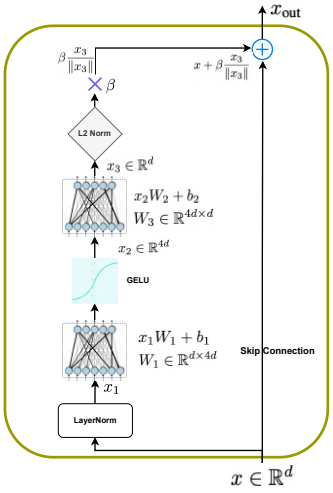
$$h_{\mathcal{N}(v)}^i = \frac{h_{\mathcal{N}(v)}^i}{\left\|h_{\mathcal{N}(v)}^i\right\|}$$

$$h_v^{i+1} = \alpha^i h_v^i + (1-\alpha^i) h_{\mathcal{N}(v)}^i$$

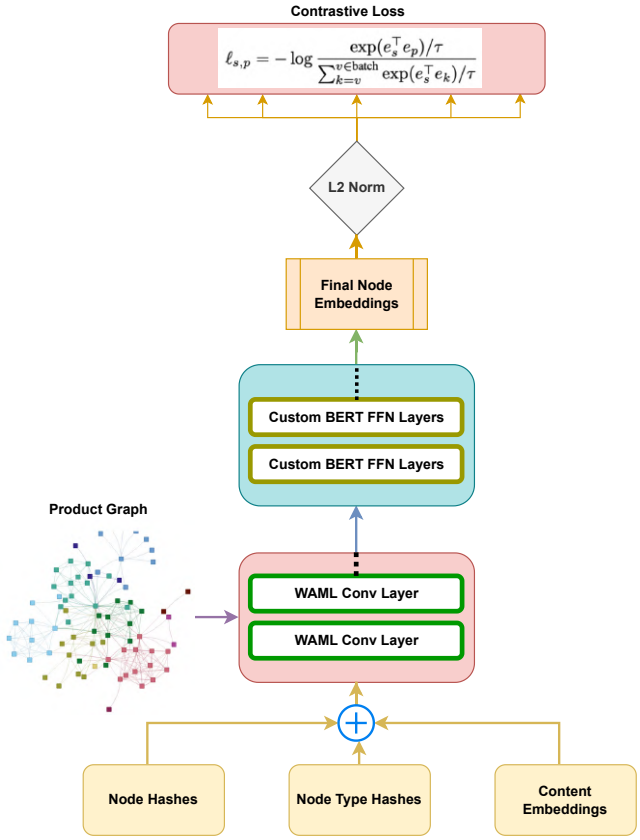
Our WAML stack is composed of K parameter free layers with only a tunable α^i per layer i and each WAML layer uses output of previous layer as input node features. Tuning α^i allows us to ensure that h_v^i is not diluted with neighbourhood features and content embeddings x_v which were part of initial node features h_v^0 remain relevant after multiple WAML layers after graph structure has been incorporated into final node embedding h_v^K. In GCMC [18], GAT [20] and GraphSAGE [7] the node and neighbourhood



(a) WAML layer



(b) Our custom BERT-FFN



(c) WAML Architecture

Figure 2: WAML architecture and it's components.

combination is learned with a neural network at each layer through back-propagation, this results in the graph structure taking high precedence over content embeddings, and overfitting to the graph structure present in training. LightGCN [8] simply takes sum of neighbourhood nodes and uses that as next layer representation of node v as $h_v^{i+1} = h^i_{\mathcal{N}(v)}$. which dilutes node content embeddings with neighbourhood features without any control in our hands.

2.3 WAML training with Contrastive loss

After we obtain node embeddings $h_v^K : \forall v \in \mathcal{V}$ from WAML stack as described in Section 2.2, we propagate them through a customised BERT-FFN network inspired by Vaswani et al. [19, Section 3.3], to obtain final node embeddings and then calculate the loss on these node embeddings. Our method uses a contrastive loss while previous methods [9] simply predicted implicit ratings. We pass h_v^K through customised BERT-FFN layer stack, see Figure 2b. Each BERT-FFN layer $j \in J$, where $J = 3$ is number of BERT-FFN layers, takes input from previous layer as $e_v^j : \forall v \in \mathcal{V}$ and produces $e_v^{j+1} : \forall v \in \mathcal{V}$ for input to next layer. Final layer outputs of node representations $e_v^J : \forall v \in \mathcal{V}$ are L2-normed and then passed onto the loss function. L2-normalizing the output vectors as $e_v = \frac{e_v^J}{\left\|e_v^J\right\|}$, ensures the loss is minimized in the cosine similarity space, these vectors can be efficiently queried using any off the shelf K-nearest neighbour search engine. Each BERT-FFN layer does the following for an input vector $x \in \mathbb{R}^d$:

$$
\begin{aligned}
x_1 &= \text{LayerNorm}(x) \\
x_2 &= \text{GELU}(x_1 W_1 + b_1) \mid W_1 \in \mathbb{R}^{d \times 4d}, x_2 \in \mathbb{R}^{4d} \\
x_3 &= x_2 W_2 + b_2 \mid W_3 \in \mathbb{R}^{4d \times d}, x_3 \in \mathbb{R}^{d} \\
x_{\text{out}} &= x + \beta \frac{x_3}{\|x_3\|}
\end{aligned}
$$

In last step BERT-FFN adds the input x to it's internal representation x_3 after normalization and multiplication by β as $x_{\text{out}} = x + \beta \frac{x_3}{\|x_3\|}$. Normalization followed by β scaling ensures that the input x is only slightly changed, which is essential to preserve content attributes x_v which were present in WAML stack's input h_v^0 and were propagated to BERT-FFN stack as input $e_v^0 = h_v^K$.

Inspired by Chen et al. [1, Section 2.1], our contrastive loss function aims to minimize the distance between node pairs $(s, p) \in E^{\mathcal{S} \cup \mathcal{P}_t}$ where $p \in \mathcal{P}_t$ and $s \in \mathcal{S}$, while maximizing distance between node pairs $(s, p) \notin E^{\mathcal{S} \cup \mathcal{P}_t}$, intuitively we minimize distance between node pairs belonging to seller-product edges $E^{\mathcal{S} \cup \mathcal{P}_t}$ to bring seller nodes close to their linked product nodes, while we maximize distance between seller and product nodes which are not

Table 3: Differences between WAML and other graph based methods.

Architecture aspect	GAT	LightGCN	WAML
Node embeddings	Trainable	Trainable	Hashing scikit
Node Type embeddings	None	None	One-hot
Content embeddings	Yes	None	Yes, from BERT
Self connection	Yes	None	Weighted
Neighbour aggregator	Attention	$\sum_{\mathcal{N}(v)} h_u^i$	$\alpha.h_v^i + (1-\alpha).h_{\mathcal{N}(v)}^i$
Trainable layer weights	Yes	None	None
Layer-wise non-linearity	RELU	None	None
Training objective	Rating prediction	BPR Rendle et al.	contrastive loss
Regularization	Dropout, L2	L2	L2, Dropout
Negative sampling	None	None	same mini-batch
Normalization per layer	$\frac{h_v^i}{\sqrt{\lvert\mathcal{N}(v)\rvert}}$	$\frac{h_v^i}{\sqrt{\lvert\mathcal{N}(v)\rvert}}$	$\frac{h_v^i}{\sqrt{\lvert\mathcal{N}(v)\rvert}}$, L2
DNN after Graph-conv	bilinear decoder	None	BERT-FFN
Output normalization	None	None	L2

linked in the graph. We randomly sample a minibatch of N seller-product edges $(s, p) \in E^{\mathcal{S}\cup\mathcal{P}_t}$ and perform contrastive matching on seller-product edges, while using the remaining nodes $2(N-1)$ in batch as negative examples. The loss function for a positive seller-product pair (s, p) whose final representations are e_s and e_p is defined as

$$\ell_{s,p} = -\log \frac{\exp(e_s^\top e_p)/\tau}{\sum_{k=v}^{v\in\text{batch}} \exp(e_s^\top e_k)/\tau} \tag{1}$$

where $\tau = 0.1$ is a temperature parameter. We take mean of this loss function over all edges (s, p) in our mini-batch as our overall loss for the batch and train WAML using back propagation. We list various architectural differences between WAML and other graph based methods in Table 3.

3 RESULTS AND ABLATIONS

We test our WAML algorithm on our productionized usecase of seller-product recommendations. The process of dataset creation is mentioned in Section 2.1 and the dataset statistics are covered in Table 5. Baseline and ablation results are based on the product graph created in Section 2.1, while we provide full product graph based results only on our final architecture. We use Recall@100 metric which determines the absolute retrieval capability for the first 100 results. For performance comparison we consider LightGCN + Content features as our primary baseline. While starting with our experiments we trained a **base** model and then made changes from base model to build our WAML model, we see the below differences to characterize our **base** architecture vs our final WAML architecture:

(1) We start with trainable node embeddings and don't use node id hashes or node type hashes. We also exclude content features x_v.
(2) We perform simple addition of node and neighbour embeddings with $\alpha^i = 0.5$.
(3) No L2 normalization anywhere, normalize neighbour embeddings by $h_{\mathcal{N}(v)}^i = \frac{h_{\mathcal{N}(v)}^i}{\sqrt{\lvert\mathcal{N}(v)\rvert}}$.
(4) No BERT-FFN layer after WAML, pass WAML output to loss function.
(5) Triplet loss as our initial loss function.

In Table 4, we present our results with ablations. Compute comparison is based on wall time on p3.2x large AWS machine (1 Nvidia V100 16GB). Memory comparison is based on maximum memory used during its execution, divided by batch size. Trainable params for the entire neural network and for non-embedding GNN/FFN parts are reported side by side, for SVD only params are the parameters of the factorized user and item matrices. The SVD implementation is taken from scikit-surprise python library [10] which has efficient C++ implementation of SVD. The remaining algorithms were implemented in python and pytorch [15]. Content features are integrated by processing them first with a simple feed forward network. We set node embeddings to 256 dimensions and content features to 256 dimensions and train using AdamW [12] optimiser with a learning rate of 0.0001, using a train-validation-test split to minimize loss on validation split and then report results on test split.

Few observations that can be inferred from our results:

(1) The majority of parameters is composed of user and product embedding matrices. We have $1.86B$ parameters for embeddings only. When we drop node embeddings and use node id hash [17] we see a drastic drop in both compute and memory usage, and a performance degradation in Recall@100 from 0.1820 to 0.1486. Node embeddings encode graph structure, addition of node id and type hash helps us to encode graph structure with no params and less compute. Unlike previous methods like GraphSAGE [7] and PinSAGE [26] which are purely inductive and skip any node identifiers for scaling, we notice that non-trainable node identifier hashes increase WAML's performance while sacrificing pure inductive capabilities which are not important to our usecase.
(2) Adding L2-norm in our architecture reduces our overall compute due to faster convergence. L2-Norms are inexpensive operation on GPU which stabilise the network training and improve performance minorly.

Model	Compute (vs Ours)	Max Memory (vs Ours)	Params (All / Network)	Recall@100	Recall@100 (Gain%)
			Baselines		
SVD	2.2x	0.4x	1.9B / 0.0M	0.0151	92.10% ↓
NCF + Content	7.5x	5.2x	1.9B / 3.2M	0.0626	67.26% ↓
GCMC	15x	9.0x	1.9B / 4.3M	0.1229	35.72% ↓
GAT + DropoutNet	26x	12.1x	1.9B / 8.2M	0.1630	14.74% ↓
LightGCN + Content	7.1x	8.2x	1.9B / 1.1M	0.1912	00.00% ↑
			Ours		
Base	5.8x	6.5x	1.9B / 0.0M	0.1391	27.25% ↓
+ Content	7.1x	7.8x	1.9B / 1.1M	0.1871	02.14% ↓
- Node embeddings	1.1x	0.7x	1.1M / 1.1M	0.1426	25.42% ↓
+ Node Id and type hash	1.3x	0.7x	1.1M / 1.1M	0.1607	15.95% ↓
+ L2-Norm WAML	1.1x	0.7x	1.1M / 1.1M	0.1721	09.98% ↓
+ tune α^i	1.1x	0.7x	1.1M / 1.1M	0.2211	15.63% ↑
+ Simple FFN	1.4x	0.8x	3.2M / 3.2M	0.2574	34.62% ↑
+ BERT-FFN	1.8x	1.0x	6.6M / 6.6M	0.2720	42.25% ↑
+ L2-Norm BERT-FFN	1.6x	1.0x	6.6M / 6.6M	0.2751	43.88% ↑
+ tune β	1.6x	1.0x	6.6M / 6.6M	0.2835	48.27% ↑
+ Contrastive loss	1.2x	1.0x	6.6M / 6.6M	0.3136	64.01% ↑
+ L2-Norm output	1.0x	1.0x	6.6M / 6.6M	0.3179	**66.27%** ↑
+ full dataset	45x	1.0x	6.6M / 6.6M	0.4051	**111.8%** ↑

Table 4: Results of various algorithms on our use-case. See Table 6 for more detailed results.

(3) Node type hashes increase performance minorly, in our case since we only have three node types but we believe with usecases having higher variety of node types we will observe more gains from node type hashes.
(4) Tuning α^i for each layer leads to huge jump in performance, a gain of 40% over **base** and 15% over LightGCN. Our network has 5 WAML layers and their $\alpha^i = [0.4, 0.45, 0.5, 0.6, 0.7]$, $i \in \{1, \ldots, 5\}$. $\alpha^i \leq 0.5 \mid i \leq 3$ implying that neighbourhood information is necessary in early layers, but $\alpha^i > 0.5 \mid i > 3$ implying node's own information is important towards the end WAML layers. Being able to tune the proportion of neighbourhood vs node's own information per layer enables us to tilt the algorithm towards content features and solve our soft cold start issue.
(5) Contrastive loss decreases our training time compared to triplet loss, since in triplet loss one node is only compared to one positive and one negative sample, but in case of contrastive loss we can use the entire batch of nodes as negative examples. This leads to faster convergence while boosting recall by 11% (0.2835 to 0.3136).
(6) Graph networks like GAT exploit graph structure and their Recall@100 is good in collaborative domain, but in our case they de-emphasise content features and provide no control over how content features can be used, resulting in low performance and high compute.
(7) LightGCN with content features outperforms GAT since content features aren't diluted through deep neural network in LightGCN, as it uses a simple addition to aggregate neighbourhood features. NCF and NCF + content both don't exploit graph structure and as a result perform lowest while still using significant compute.
(8) Using the non-filtered full dataset provides modest recall improvement of 27% (0.3179 to 0.4051) over the filtered dataset with our WAML method but also results in massively increased compute by **45**x for training which proves the efficacy of our filtering method in Section 2.1.

4 CONCLUSION

We encountered a novel recommendation system problem where we have a perpetual soft item cold start issue for all candidate recommendation items. Our items/products always have very few interactions and once they gather enough interactions they are removed from the candidate set. We recognised that e-commerce recommendation system problems can be viewed as a link prediction problem in a partially observed dynamic graph. Traditional algorithms like collaborative filtering with matrix factorization and more modern algorithms such as graph neural networks are meant to work where items have high number of interactions with content features used for only assisting the CF algorithm. Content based approaches on the other hand completely rely on content features and ignore the product graph. Simultaneously, it has been observed that graph neural networks are low pass filters on the graph data structure [14], and for recommendation systems, feature transformations and non-linear activation on each graph neural network layer has no positive effect on collaborative filtering [8]. These findings correlate with our findings and show why complex networks like GAT and GCMC underperform compared to WAML and LightGCN.

A controlled approach to combining content with product graph performs best in our domain. We also notice the recall to compute trade-off in modern graph algorithms such as GAT and GCMC is sub-par for recommendation systems, these algorithms are more suited for node and graph classification/regression tasks [8]. LightGCN, a graph algorithm especially enhanced for recommendation systems provides a good compute balance but still fails to scale to

production use cases and solve our soft cold start issue. We propose several changes, removal of node embeddings, graph reduction mechanism and controlled combination between node and neighbourhood embeddings through our WAML method in Sec. 2 which enable us to scale and provide high recall for our use-case. Our innovative graph reduction mechanism reduces product graph size from $\geq 5B$ edges, $400M$ nodes to just $20M$ (250x lower) edges, $6M$ (66x lower) nodes[4]. , resulting in 45x reduction in compute while retaining 79.5% of maximum recall, see Table 4. On the same graph size our WAML method provides a gain of **66.27%** over next best method by LightGCN.

REFERENCES

[1] Ting Chen, Simon Kornblith, Mohammad Norouzi, and Geoffrey Hinton. 2020. A Simple Framework for Contrastive Learning of Visual Representations. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, 1597–1607. https://proceedings.mlr.press/v119/chen20j.html

[2] Ting Chen, Simon Kornblith, Kevin Swersky, Mohammad Norouzi, and Geoffrey E Hinton. 2020. Big Self-Supervised Models are Strong Semi-Supervised Learners. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 22243–22255. https://proceedings.neurips.cc/paper/2020/file/fcbc95ccdd551da181207c0c1400c655-Paper.pdf

[3] J. Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT*. https://arxiv.org/pdf/1810.04805.pdf

[4] Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. 2019. Graph Neural Networks for Social Recommendation. In *The World Wide Web Conference* (San Francisco, CA, USA) *(WWW '19)*. Association for Computing Machinery, New York, NY, USA, 417–426. https://doi.org/10.1145/3308558.3313488

[5] Chen Gao, Yu Zheng, and Nian Li. 2021. Graph Neural Networks for Recommender Systems: Challenges, Methods, and Directions. https://arxiv.org/abs/2109.12843

[6] Huifeng Guo, Ruiming TANG, Yunming Ye, Zhenguo Li, and Xiuqiang He. 2017. DeepFM: A Factorization-Machine based Neural Network for CTR Prediction. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*. 1725–1731. https://doi.org/10.24963/ijcai.2017/239

[7] William L. Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 1024–1034. https://proceedings.neurips.cc/paper/2017/hash/5dd9db5e033da9c6fb5ba83c7a7ebea9-Abstract.html

[8] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, YongDong Zhang, and Meng Wang. 2020. *LightGCN: Simplifying and Powering Graph Convolution Network for Recommendation.* Association for Computing Machinery, New York, NY, USA, 639–648. https://doi.org/10.1145/3397271.3401063

[9] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural Collaborative Filtering. In *Proceedings of the 26th International Conference on World Wide Web* (Perth, Australia) *(WWW '17)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 173–182. https://doi.org/10.1145/3038912.3052569

[10] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (2020), 2174. https://doi.org/10.21105/joss.02174

[11] Yehuda Koren, Robert Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37. https://doi.org/10.1109/MC.2009.263

[12] Ilya Loshchilov and Frank Hutter. 2017. Fixing Weight Decay Regularization in Adam. *CoRR* abs/1711.05101 (2017). arXiv:1711.05101 http://arxiv.org/abs/1711.05101

[13] Xin Luo, Mengchu Zhou, Yunni Xia, and Qingsheng Zhu. 2014. An Efficient Non-Negative Matrix-Factorization-Based Approach to Collaborative Filtering for Recommender Systems. *IEEE Transactions on Industrial Informatics* 10, 2 (2014), 1273–1284. https://doi.org/10.1109/TII.2014.2308433

[14] Hoang NT and Takanori Maehara. 2019. Revisiting Graph Neural Networks: All We Have is Low-Pass Filters. https://doi.org/10.48550/ARXIV.1905.09550

[15] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[16] Steffen Rendle, Christoph Freudenthaler, Zeno Gantner, and Lars Schmidt-Thieme. 2009. BPR: Bayesian Personalized Ranking from Implicit Feedback. In *UAI 2009, Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence, Montreal, QC, Canada, June 18-21, 2009*, Jeff A. Bilmes and Andrew Y. Ng (Eds.). AUAI Press, 452–461. https://dslpitt.org/uai/displayArticleDetails.jsp?mmnu=1&smnu=2&article_id=1630&proceeding_id=25

[17] scikit. 2022. HashingVectorizer. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html [Online].

[18] Rianne van den Berg, Thomas N. Kipf, and Max Welling. 2017. Graph Convolutional Matrix Completion. *CoRR* abs/1706.02263 (2017). arXiv:1706.02263 http://arxiv.org/abs/1706.02263

[19] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[20] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rJXMpikCZ

[21] Maksims Volkovs, Guangwei Yu, and Tomi Poutanen. 2017. DropoutNet: Addressing Cold Start in Recommender Systems. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/dbd22ba3bd0df8f385bdac3e9f8be207-Paper.pdf

[22] Jizhe Wang, Pipei Huang, Huan Zhao, Zhibo Zhang, Binqiang Zhao, and Dik Lun Lee. 2018. Billion-scale Commodity Embedding for E-commerce Recommendation in Alibaba. arXiv:arXiv:1803.02349 https://arxiv.org/abs/1803.02349

[23] Shoujin Wang, Liang Hu, Yan Wang, Xiangnan He, Quan Z. Sheng, Mehmet A. Orgun, Longbing Cao, Francesco Ricci, and Philip S. Yu. 2021. Graph Learning based Recommender Systems: A Review. In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI-21*, Zhi-Hua Zhou (Ed.). International Joint Conferences on Artificial Intelligence Organization, 4644–4652. https://doi.org/10.24963/ijcai.2021/630 Survey Track.

[24] Yinwei Wei, Xiang Wang, Qi Li, Liqiang Nie, Yan Li, Xuanping Li, and Tat-Seng Chua. 2021. Contrastive Learning for Cold-Start Recommendation. *CoRR* abs/2107.05315 (2021). arXiv:2107.05315 https://arxiv.org/abs/2107.05315

[25] Tiansheng Yao, Xinyang Yi, Derek Zhiyuan Cheng, Felix Yu, Ting Chen, Aditya Menon, Lichan Hong, Ed H. Chi, Steve Tjoa, Jieqi (Jay) Kang, and Evan Ettinger. 2021. *Self-Supervised Learning for Large-Scale Item Recommendations.* Association for Computing Machinery, New York, NY, USA, 4321–4330. https://doi.org/10.1145/3459637.3481952

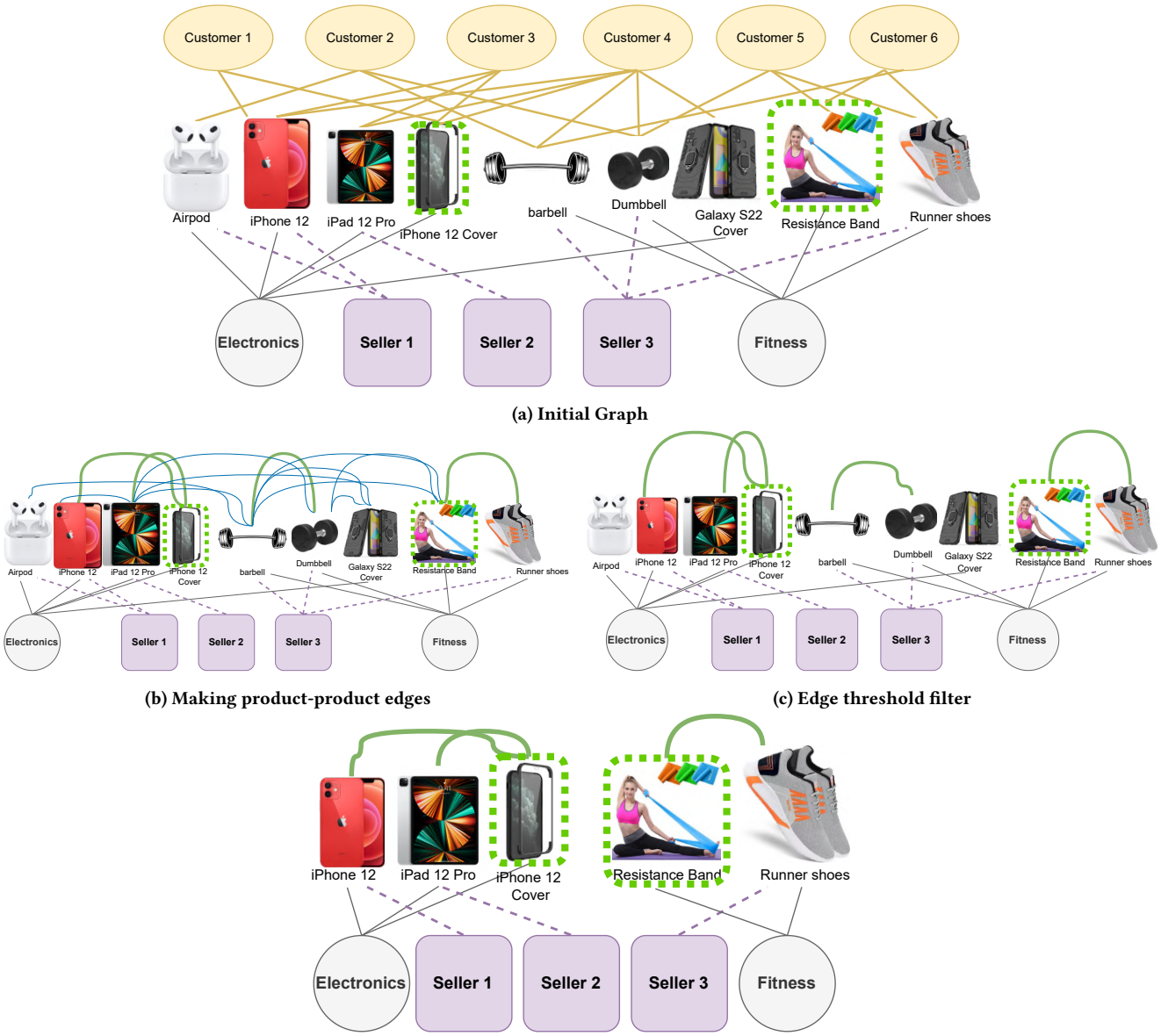
[26] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. 2018. Graph Convolutional Neural Networks for Web-Scale Recommender Systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, KDD 2018, London, UK, August 19-23, 2018*, Yike Guo and Faisal Farooq (Eds.). ACM, 974–983. https://doi.org/10.1145/3219819.3219890

[27] Yuefeng Zhang. 2022. An Introduction to Matrix factorization and Factorization Machines in Recommendation System, and Beyond. https://doi.org/10.48550/ARXIV.2203.11026

A BUILDING THE PRODUCT GRAPH

Within any e-commerce store we have a set of customers C, products $\mathcal{P}$ and sellers $\mathcal{S}$ selling these products. Let $|C|$ be the total number of customers, $|\mathcal{P}|$ be total products and $|\mathcal{S}|$ be total sellers. A customer $c_i \in C$ with $i \in \{1, \ldots, |C|\}$ interacts with a product $p_j \in \mathcal{P}$ with $j \in \{1, \ldots, |\mathcal{P}|\}$ through clicks, add to cart, views and purchases forming an edge tuple as $(c_i, p_j) \in E^{C \cup \mathcal{P}}$ where $E^{C \cup \mathcal{P}}$ denotes all customer-product edges. Similarly a seller

[4]Numbers given here are for demonstrative purposes to give a sense of how our method down-samples the product graph. These are not actual business derived numbers of any e-commerce store.



(a) Initial Graph

(b) Making product-product edges

(c) Edge threshold filter

(d) Remove products not connected to candidate set $\mathcal{P}_r$

Figure 3: Process to reduce edges and nodes in graph. Products surrounded in green dotted boxes belong to our candidate set $\mathcal{P}_r$. (a) Initial Graph with all customers C and products $\mathcal{P}$. (b) Remove customers and connect products interacted by same customer. (c) Remove product-product edges below a fixed threshold, here threshold = 2. (d) Remove products not connected to candidate set $\mathcal{P}_r$, remaining products form our training product set $\mathcal{P}_t | \mathcal{P}_r \subset \mathcal{P}_t$.

$s_k \in \mathcal{S}$ with $k \in \{1, \ldots, |\mathcal{S}|\}$ interacts with product p_j by listing/offering the product for sale in the store forming an edge tuple as $(s_k, p_j) \in E^{\mathcal{S} \cup \mathcal{P}}$ where $E^{\mathcal{S} \cup \mathcal{P}}$ denotes all seller-product edges.

Apart from customer-product and seller-product edges, each product is also associated with its corresponding product category $a_m \in \mathcal{A}$ with $m \in \{1, \ldots, |\mathcal{A}|\}$, where $\mathcal{A}$ denotes all product categories in the store with $|\mathcal{A}|$ total categories. This product p_j to product-category a_m mapping produces edge tuples as $(p_j, a_m) \in E^{\mathcal{A} \cup \mathcal{P}}$ where $E^{\mathcal{A} \cup \mathcal{P}}$ denotes all product to product category edges.

As a result, the overall set of entities and their interactions can be represented as an undirected graph as shown in Figure 3a.

After obtaining customers C, products $\mathcal{P}$, sellers $\mathcal{S}$ and customer-product edges $E^{C \cup \mathcal{P}}$, seller-product edges $E^{\mathcal{S} \cup \mathcal{P}}$ and product to product category edges $E^{\mathcal{A} \cup \mathcal{P}}$. While the count of set of products $\mathcal{P}$ is $|\mathcal{P}| > 200$ Million, the count $|\mathcal{P}_r|$ of our candidate set of products $\mathcal{P}_r \subset \mathcal{P}$ from which we recommend to sellers is $|\mathcal{P}_r| \sim 500K$. Also our recommendation system is intended for sellers only. With these two observations, we decided that we can remove all customers

Table 5: Graph edge and node counts before and after our filtering technique.

Symbol	Description	Approximate Count
$\mathcal{S}$	Sellers	$1x$
$\mathcal{C}$	Customers	$1 \times 10^3 x$
$\mathcal{P}$	Products	$1 \times 10^3 x$
$\mathcal{A}$	Product categories	≤ 100
$\mathcal{P}_\mathrm{r}$	Candidate Products	$5x$
$\mathcal{P}_\mathrm{t}$	Training product set	$30x$
$E^{\mathcal{C}\cup\mathcal{P}}$	Customer-Product Edges	$2 \times 10^4 x$
$E^{\mathcal{P}}$	Unfiltered product-product edges	$1 \times 10^4 x$
$E^{\mathcal{S}\cup\mathcal{P}}$	Unfiltered seller-product edges	$2 \times 10^2 x$
$E^{\mathcal{A}\cup\mathcal{P}}$	Unfiltered product to product categories edges	$1 \times 10^2 x$
$E^{\mathcal{P}_\mathrm{t}}$	Filtered product-product edges	$80x$
$E^{\mathcal{S}\cup\mathcal{P}_\mathrm{r}}$	Seller to candidate product edges	$2.2x$
$E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}}$	Seller to filtered product edges	$10x$
$E^{\mathcal{A}\cup\mathcal{P}_\mathrm{t}}$	Filtered product to product categories edges	$30x$
$\mathcal{V} = \mathcal{S} \cup \mathcal{P}_\mathrm{t} \cup \mathcal{A}$	Final Graph Nodes	$31x$
$\mathcal{E} = E^{\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{A}\cup\mathcal{P}_\mathrm{t}}$	Final Graph Edges	$120x$

C nodes as well as any product node not in 1-hop neighbourhood of any node in $\mathcal{P}_\mathrm{r}$. Specifically we followed the below steps for node and edge filtering, an example of such process can be seen in Figure 3.

(1) For any two customer-product edge (c_i, p_j) and (c_i, p_k) where customer c_i is common, we add a new product-product edge $(p_j, p_k) \in E^{\mathcal{P}}$ where $E^{\mathcal{P}}$ denote all product-product edges. Then We remove all customer nodes C and all customer-product edges $E^{\mathcal{C}\cup\mathcal{P}}$
(2) There can be multiple edges between two products p_i and p_k if both products were interacted with together by multiple customers. For example, p_i can be "iPhone 12 Pro 256 GB" and p_j can be "iPhone 12 Pro case cover". Some irrelevant product-product pairs may also be included with low frequency, to capture only relevant pairs we apply a threshold of 200 and drop any pair (p_j, p_k) which has occurred less times than our threshold.
(3) We consider products in our candidate set $\mathcal{P}_\mathrm{r} \subset \mathcal{P}$ and keep only those product-product edges (p_j, p_k) where $p_j \in \mathcal{P}_\mathrm{r}$ or $p_k \in \mathcal{P}_\mathrm{r}$. Intuitively, one of the product node in a product-product edge must be in our smaller candidate set of recommendations. This gives us a new product subset which includes the candidate set $\mathcal{P}_\mathrm{r}$ and few other products from $\mathcal{P}$, which we refer as our training product set $\mathcal{P}_\mathrm{t} \mid \mathcal{P}_\mathrm{r} \subset \mathcal{P}_\mathrm{t} \subset \mathcal{P}$ with its count $|\mathcal{P}_\mathrm{t}| : |\mathcal{P}_\mathrm{r}| \ll |\mathcal{P}_\mathrm{t}| \ll |\mathcal{P}|$. This reduces the count of product-product edges giving us $E^{\mathcal{P}_\mathrm{t}} \subset E^{\mathcal{P}}$ as our final product-product edges.
(4) We keep all sellers $\mathcal{S}$ and any seller-product edges (s_k, p_j) where $p_j \in \mathcal{P}_\mathrm{t}$ with $j \in \{1, \ldots, |\mathcal{P}_\mathrm{t}|\}$ to form new set of seller-product edges $E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}} \subset E^{\mathcal{S}\cup\mathcal{P}}$. Note that the number of seller to candidate product edges $E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}}$ are far less since $|\mathcal{P}_\mathrm{r}| \ll |\mathcal{P}_\mathrm{t}|$. Refer Table 1 to see how few interactions products from our candidate set $\mathcal{P}_\mathrm{r}$ get.
(5) we keep all product to product category edges (p_j, a_m) where $p_j \in \mathcal{P}_\mathrm{t}$ with $j \in \{1, \ldots, |\mathcal{P}_\mathrm{t}|\}$ to form new set of product to product category edges $E^{\mathcal{A}\cup\mathcal{P}_\mathrm{t}} \subset E^{\mathcal{A}\cup\mathcal{P}}$.

Our final graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is composed of nodes $\mathcal{V} = \mathcal{S} \cup \mathcal{P}_\mathrm{t} \cup \mathcal{A}$ and edges $\mathcal{E} = E^{\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{S}\cup\mathcal{P}_\mathrm{t}} \cup E^{\mathcal{A}\cup\mathcal{P}_\mathrm{t}}$, similar to Figure 3d. When we describe our WAML algorithm, we refer to this final graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ and train all our experiments on this graph. Note that our product graph has more products $\mathcal{P}_\mathrm{t}$ than the candidate set $\mathcal{P}_\mathrm{r}$ which helps in creating links between products from $\mathcal{P}_\mathrm{r}$ and sellers $\mathcal{S}$ since products from candidate set themselves have very less direct links with sellers. Table 5 provides details on approximate relative count of edges and nodes we obtain before and after filtering.

B EXTENDED RESULTS

We list detailed ablations and more algorithms tested with results in Table 6

Model	Compute (vs Ours)	Max Memory (vs Ours)	Params (All)	Recall@100 (Production)	Recall@100 (Offline)	Recall@100 (Gain%)
			Baselines			
Top-N	0.0x	0.0x	0.0M / 0.0M	0.002	0.0024	98.74% ↓
SVD	2.2x	0.4x	1.9B / 0.0M	0.014	0.0151	92.10% ↓
NCF	6.0x	4.0x	1.9B / 2.1M	0.021	0.0203	89.38% ↓
+ Content	7.5x	5.2x	1.9B / 3.2M	0.070	0.0626	67.26% ↓
GCMC	15x	9.0x	1.9B / 4.3M	-	0.1229	35.72% ↓
GAT	24x	12.0x	1.9B / 8.2M		0.1329	30.42% ↓
+ DropoutNet	26x	12.1x	1.9B / 8.2M	0.1529	0.1630	14.74% ↓
LightGCN	5.2x	6.1x	1.9B / 0.0M	-	0.1016	46.88% ↓
+ Content	7.1x	8.2x	1.9B / 1.1M	-	0.1912	00.00% ↑
			Ours			
Base	5.8x	6.5x	1.9B / 0.0M	-	0.1391	27.25% ↓
+ Content	7.1x	7.8x	1.9B / 1.1M	0.1820	0.1871	02.14% ↓
- Node embeddings	1.1x	0.7x	1.1M / 1.1M	0.1486	0.1426	25.42% ↓
+ Node Id hash	1.2x	0.7x	1.1M / 1.1M	0.1611	0.1591	16.78% ↓
+ Node type hash	1.3x	0.7x	1.1M / 1.1M	-	0.1607	15.95% ↓
+ L2-Norm WAML	1.1x	0.7x	1.1M / 1.1M	-	0.1721	09.98% ↓
+ tune α^i	1.1x	0.7x	1.1M / 1.1M	0.2177	0.2211	15.63% ↑
+ Simple FFN	1.4x	0.8x	3.2M / 3.2M	-	0.2574	34.62% ↑
+ BERT-FFN	1.8x	1.0x	6.6M / 6.6M	-	0.2720	42.25% ↑
+ L2-Norm BERT-FFN	1.6x	1.0x	6.6M / 6.6M	0.2804	0.2751	43.88% ↑
+ tune β	1.6x	1.0x	6.6M / 6.6M	0.2825	0.2835	48.27% ↑
+ Contrastive loss	1.2x	1.0x	6.6M / 6.6M	-	0.3136	64.01% ↑
+ L2-Norm output	1.0x	1.0x	6.6M / 6.6M	0.3116	0.3179	66.27% ↑
			Full vs Filtered dataset of Sec. 2.1			
+ WAML	1.0x	1.0x	6.6M / 6.6M	0.3116	0.3179	66.27% ↑
+ full dataset	45x	1.0x	6.6M / 6.6M	-	0.4051	**111.8%** ↑

Table 6: Results of various algorithms on our use-case.