Collaborative Metric Learning

Cheng-Kang Hsieh, Longqi Yang, Yin Cui, Tsung-Yi Lin, Serge Belongie,Deborah Estrin

박진혁

2023.07.14

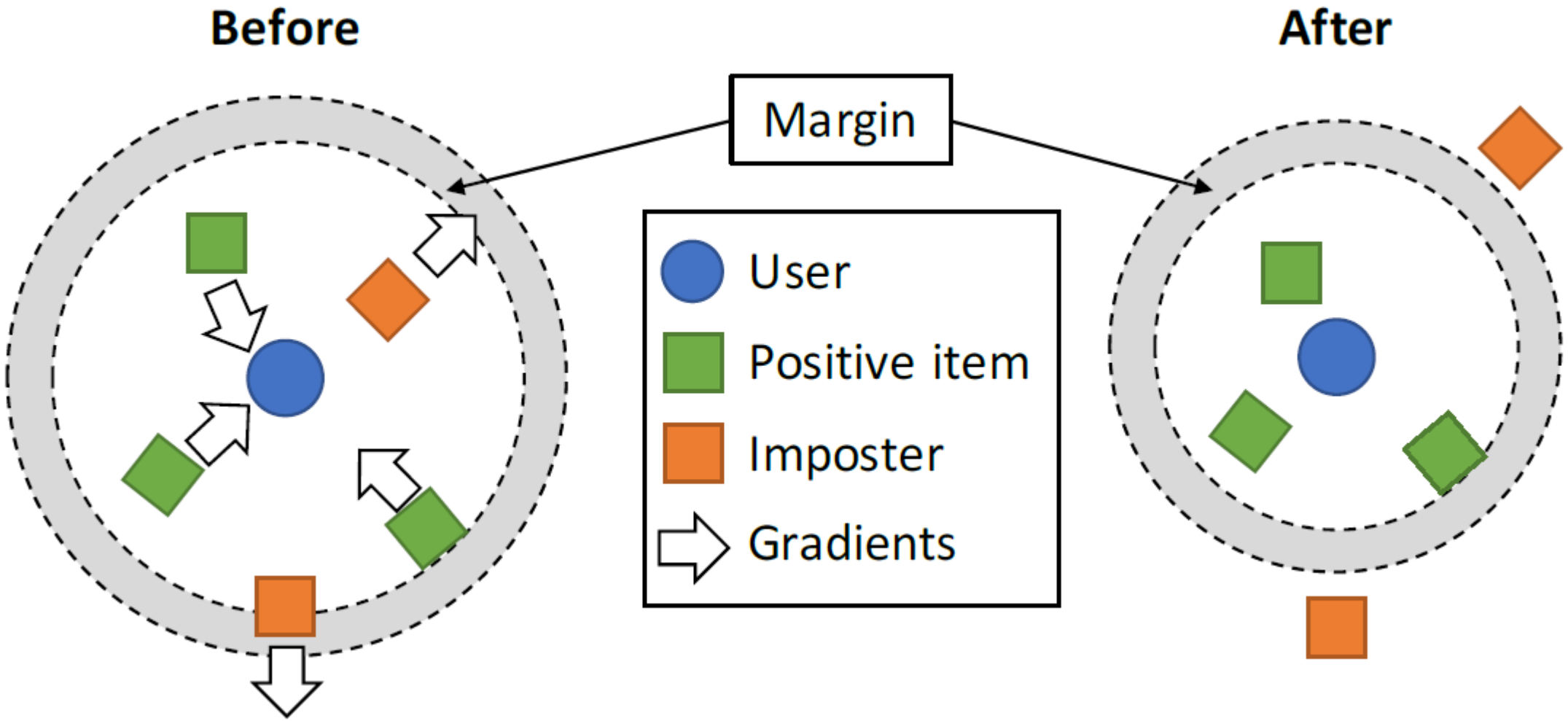
CONTENTS

1. Introduction
2. Back Ground
3. Model description
4. Evaluation
5. Conclusion
6. Review

Introduction

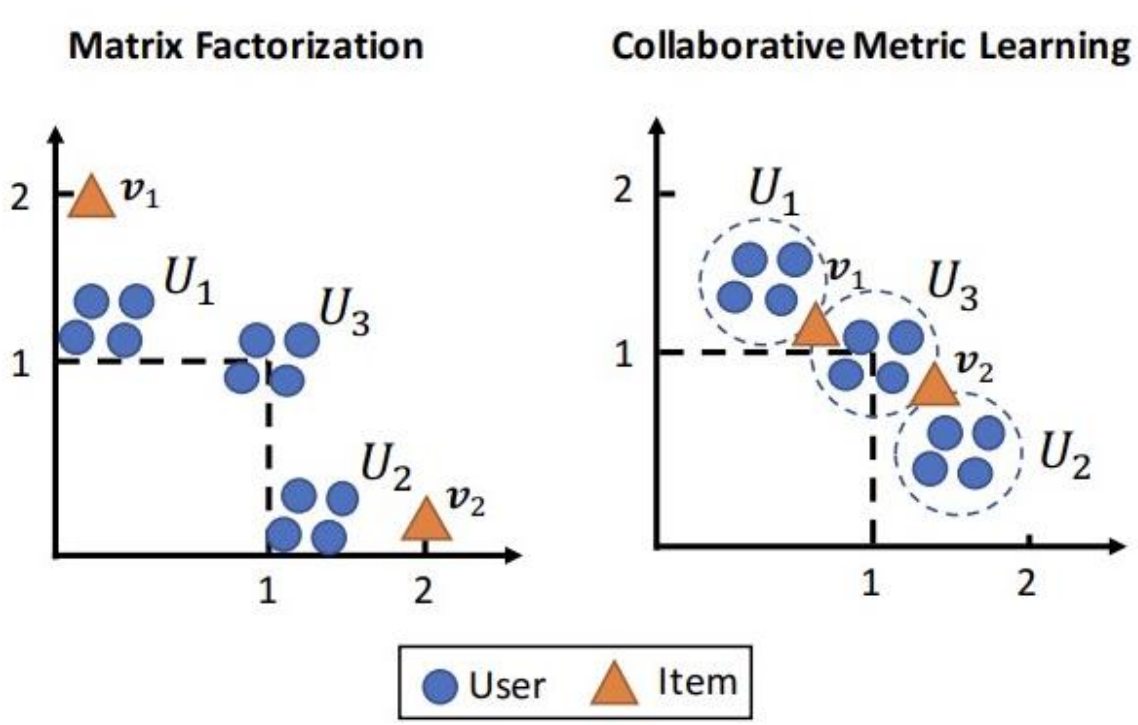
Matrix Factorization에서 발생하는 문제점(dot product does not satisfy the crucial triangle inequality)을 해결하기 위해 Collaborative Filtering + Metric Learning을 제시하는 논문

Introduction



Background

Triangle inequality



$$\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$$

1. dot product does not satisfy T.I
Ex: A = [1,1], B=[0,0], C=[100,100]
A*C < A*B + B*C -> 100+ 100 < 0

2. U3 like v1, v2 both but there is no reliably capture in item-item(user-user) similarity

3. User preference

Fig.3 Example of latent vector assignment for MF and CML

Background

Distance Metric learning & metric

1. $D(\vec{x}_i,\vec{x}_j)+D(\vec{x}_j,\vec{x}_k) \geq D(\vec{x}_i,\vec{x}_k)$ (triangular inequality).
2. $D(\vec{x}_i,\vec{x}_j) \geq 0$ (non-negativity).
3. $D(\vec{x}_i,\vec{x}_j) = D(\vec{x}_j,\vec{x}_i)$ (symmetry).
4. $D(\vec{x}_i,\vec{x}_j) = 0 \iff \vec{x}_i = \vec{x}_j$ (distinguishability).

- 1 : triangular inequality로 삼각형에서 두 변의 길이를 더하면 다른 1개 변과 길이가 같거나 크다는 점입니다.
- 2 : non-negativity로 i 벡터에서 j 벡터까지의 거리는 0보다 크거나 같습니다.
- 3 : symmetry로 metric의 input으로 두 벡터를 넣을 때 순서를 바꿔도 결과는 같습니다.
- 4 : distinguishability로 두 벡터 간 거리가 0이면 두 벡터는 같은 벡터임을 말합니다.

- pseudometric : 위의 1~3의 조건을 만족하는 경우

Background

Metric Learning

$$\min_{A} \sum_{(x_i,x_j)\in\mathcal{S}} d_A(x_i,x_j)^2$$

$$\text{s.t.} \quad \sum_{(x_i,x_j)\in\mathcal{D}} d_A(x_i,x_j)^2 \geq 1 \text{ and } A \succeq 0.$$

$$\mathcal{S} = \{(x_i, x_j) | x_i \text{ and } x_j \text{are considered similar}\},$$

$$\mathcal{D} = \{(x_i, x_j) | x_i \text{ and } x_j \text{are considered dissimilar}\}.$$

$$d_A(x_i,x_j) = \sqrt{(x_i - x_j)^T A (x_i - x_j)} \ ,$$

A is PSD(positive semi definite) -> convex matrix

1. object: 가까운건 minimize 할것
2. St: 먼것은 1이상의 distanc를 가질 것
3. A는 PSD일 것

So d_a is also convex -> convex program

⇒ This is not always feasible(답이 존재하지 않을 수도 있다)

Background

LMNN(metric learning for KNN)

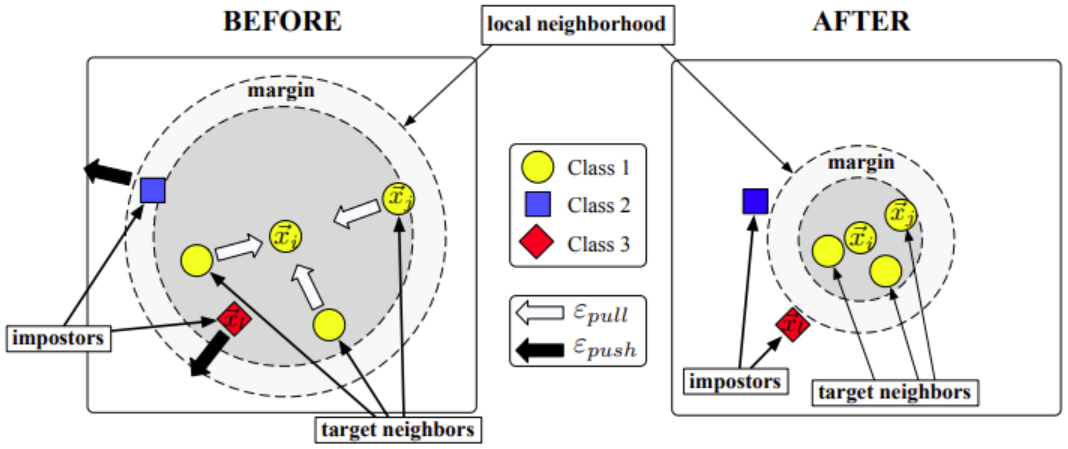


Figure 1: Schematic illustration of one input's neighborhood before training (*left*) versus after training (*right*). The distance metric is optimized so that: (i) its $k=3$ target neighbors lie within a smaller radius after training; (ii) differently labeled inputs lie outside this smaller radius by some finite margin. Arrows indicate the gradients on distances arising from different terms in the cost function.

$$\mathcal{L}_{pull}(d) = \sum_{j \rightsquigarrow i} d(x_i, x_j)^2,$$

$$\mathcal{L}_{push}(d) = \sum_{i,j \rightsquigarrow i} \sum_{k} (1 - y_{ik})[1 + d(x_i, x_j)^2 - d(x_i, x_k)^2]_+,$$

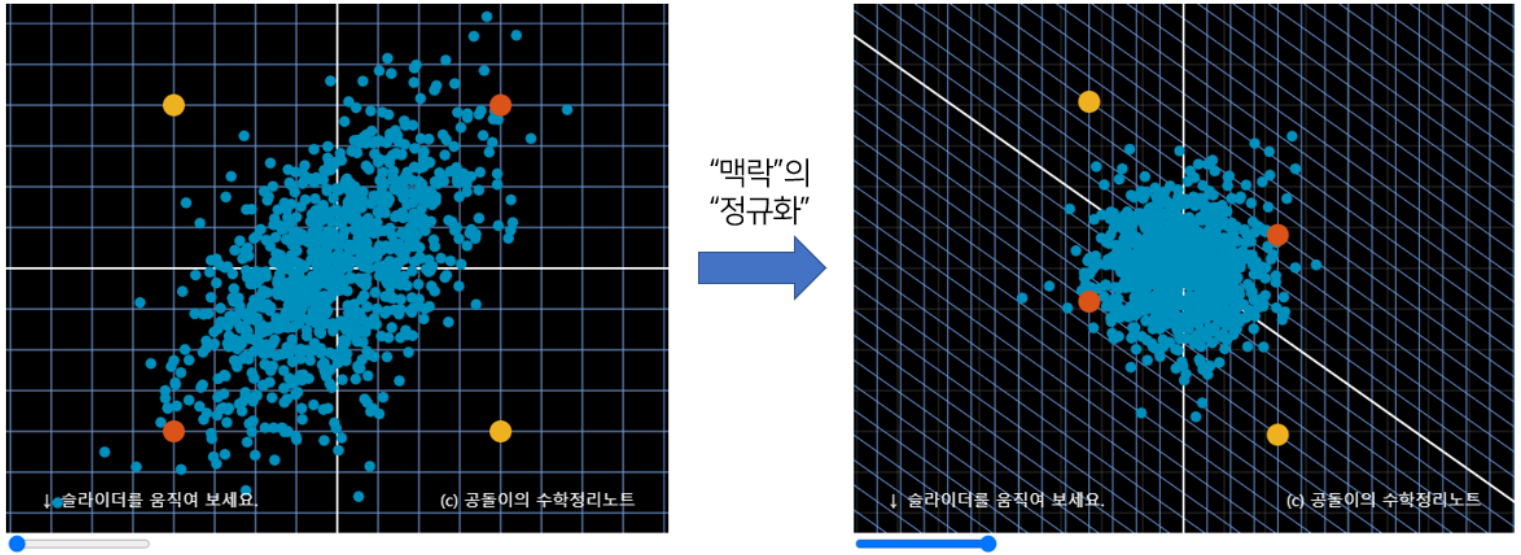
$$\boldsymbol{\varepsilon}(\mathbf{L}) = (1-\mu)\,\boldsymbol{\varepsilon}_{pull}(\mathbf{L}) + \mu\boldsymbol{\varepsilon}_{push}(\mathbf{L}).$$

Euclidean distance를 leaner transform 시키면 Mahalanobis 형태로 바뀌는 것이죠. Gaussian 분포의 quadratic form이기도 함 -> 이를 최적화하는 M(L^*L)을 찾는 것이 목적

On the other hand,Weinberger et al. showed that if the learned metric is to be used for k-nearest neighbor classication, it is sucient to just learn a metric that makes each object's k-nearest neighbors be the objects that share the same class label with that object

Background

Mahalanobis distance



$$\sqrt{(x-y)\Sigma^{-1}(x-y)^T}$$

맥락을 고려한 상대적인 거리

$$d_z = \sqrt{zz^T} = \sqrt{(xR^{-1})(xR^{-1})^T}$$

$$= \sqrt{xR^{-1}(R^{-1})^T x^T} = \sqrt{x(R^T R)^{-1} x^T} = \sqrt{x\Sigma^{-1}x^T}$$

Background

Collaborative Filtering & Matrix Factorization

$$\min_{\mathbf{u}_*,\mathbf{v}_*} \sum_{r_{ij}\in\mathcal{K}} (r_{ij} - \mathbf{u}_i^T\mathbf{v}_j)^2 + \lambda_u\|\mathbf{u}_i\|^2 + \lambda_v\|\mathbf{v}_i\|^2,$$

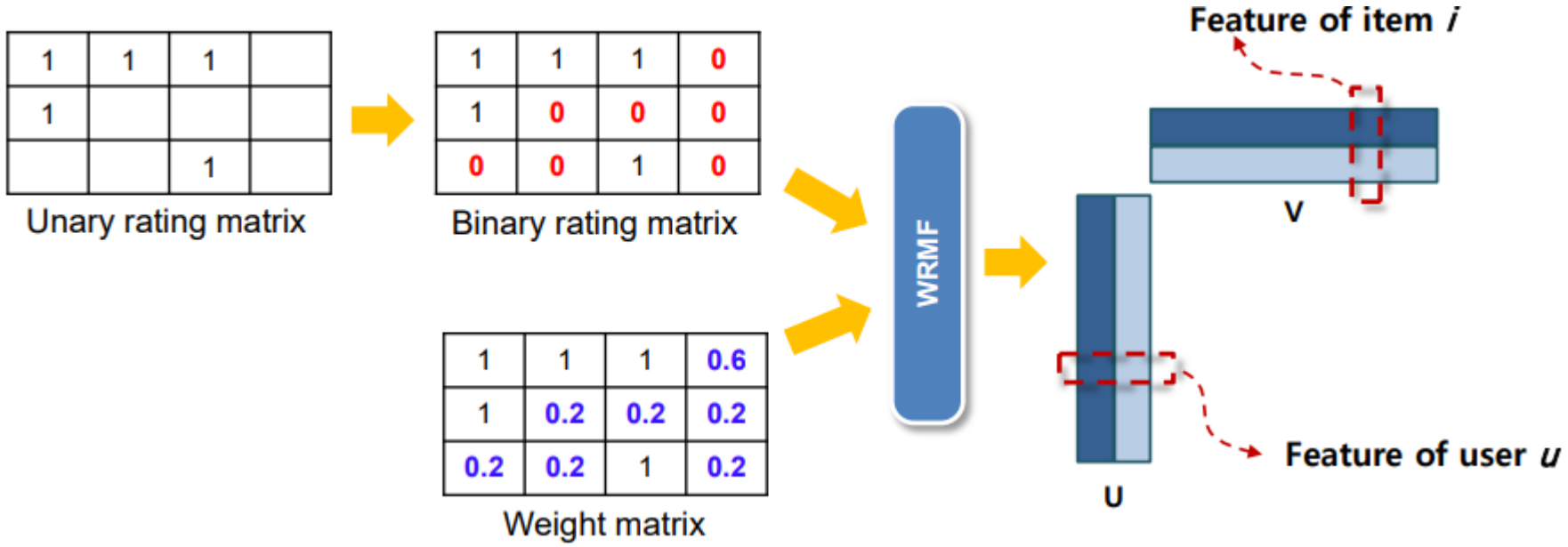
Point - wise vs Pair - wise

point-wise : 사용자의 아이템별 실제 스코어와 예측 스코어간 error를 최소화하는 목적함수를 사용

pair-wise : 사용자의 rated item과 unrated item간의 예측 스코어 차이를 극대화하는 목적함수를 사용

Background

Implicit Feedback(weighted regularized matrix factorization (WRMF))

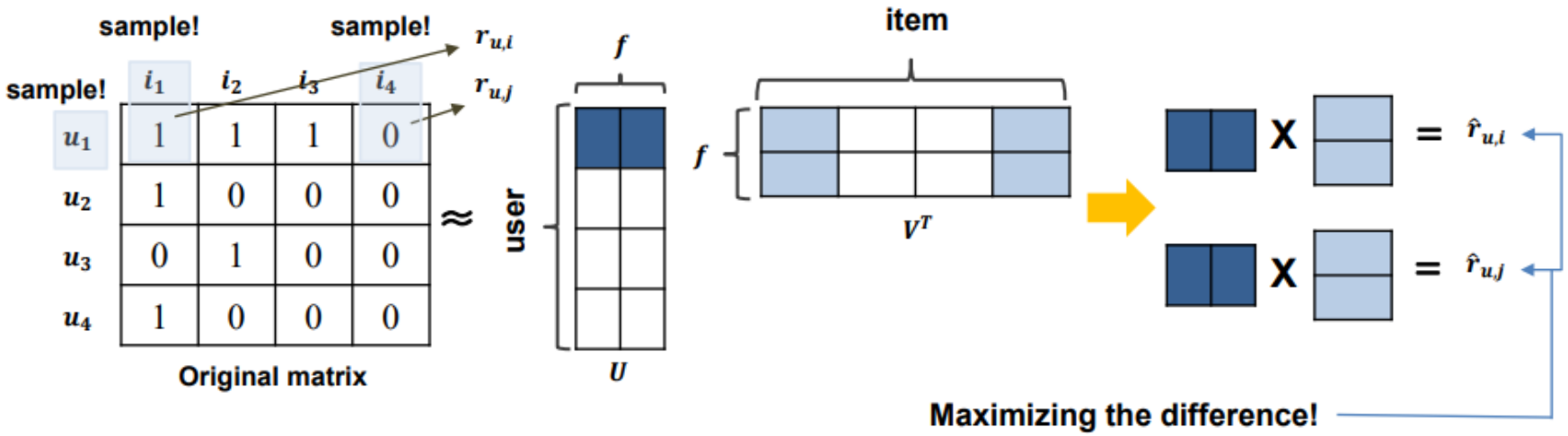


WRMF는 Unrated item을 negative preference로 가정하고 단순하게 값을 0으로 매기고, rating matrix외에 weight matrix를 생성한다.
Weight matrix란 rating matrix 값에 대한 신뢰도(confidendce) 값으로 만들어진다(관찰 된 것은 confidence값이 크다)

Background

Bayesian Personalized ranking(pair-wise)

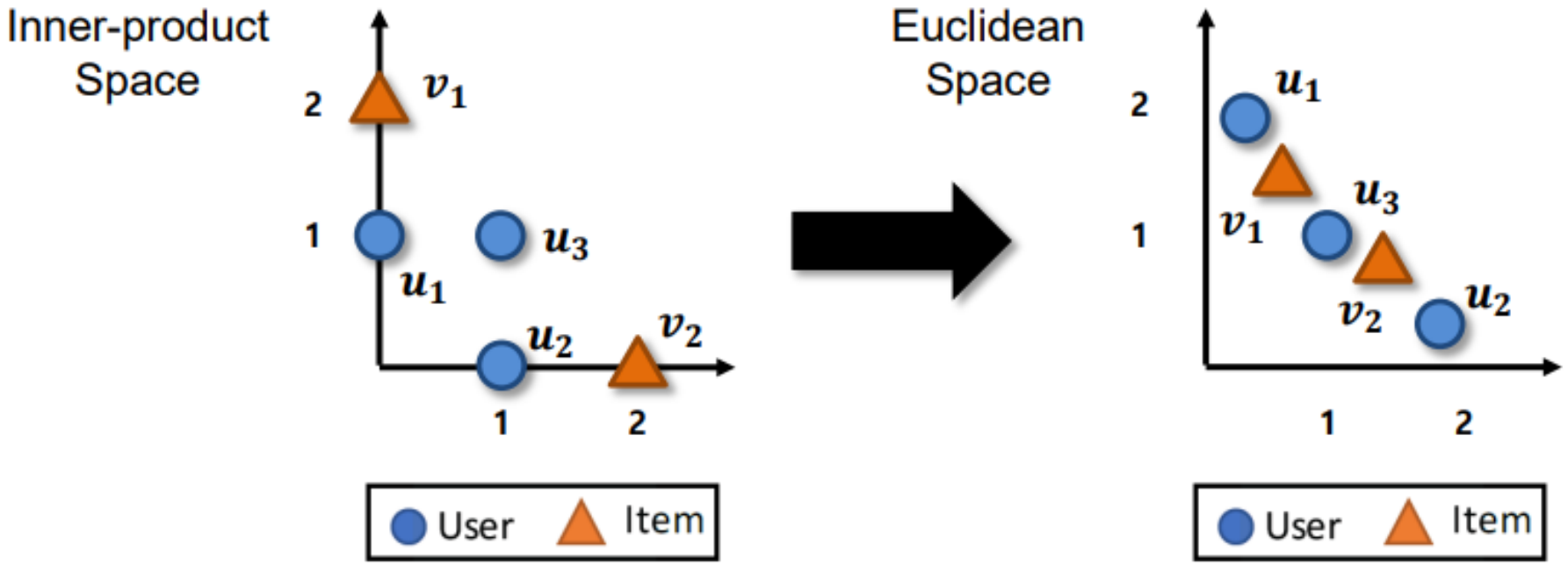
- Equation: $\underset{\theta}{\operatorname{argmax}} \sum_{u \in U} \sum_{i \in I_u^+} \sum_{j \in I_u^-} \sigma(\hat{r}_{ui} - \hat{r}_{uj})$



목적함수 식에서 +는 산 아이템, -는 안 산 아이템을 의미하는 notation이다.
즉 Rated Item과 Unrated item 두 스코어 차이에 대해 극대화하는 방식이 BPR이다.

CML Model description

목적: Metric공간에 풀어낸 아이템과 유저 간의 euclidean 거리를 최적화하는 것이 Metric learning의 학습 방향(implicit data)



no longer about estimating a specific rating matrix but about capturing users' relative preferences for different items.

CML Model description

Loss function(Total loss) & Model feature

$$\min_{\theta,\mathbf{u}_*,\mathbf{v}_*} \mathcal{L}_m + \lambda_f \mathcal{L}_f + \lambda_c \mathcal{L}_c$$

$$\text{s.t.} \quad \|\mathbf{u}_*\|^2 \leq 1 \text{ and } \|\mathbf{v}_*\|^2 \leq 1,$$

$\mathcal{L}_m$: Embedding Loss

$\mathcal{L}_f$: Feature Loss

$\mathcal{L}_c$: Covariance Loss

Model description

Embedding Loss

$$\mathcal{L}_m(d) = \sum_{(i,j)\in\mathcal{S}} \sum_{(i,k)\notin\mathcal{S}} w_{ij}[m + d(i,j)^2 - d(i,k)^2]_+, \quad (1)$$

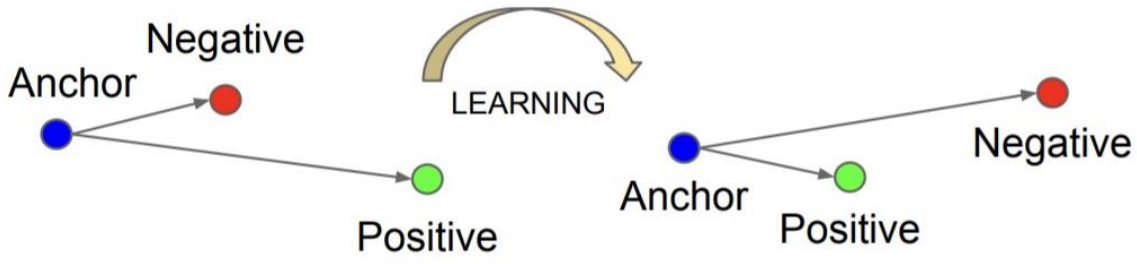
$$d(i,j) = \|\mathbf{u}_i - \mathbf{v}_j\|,$$

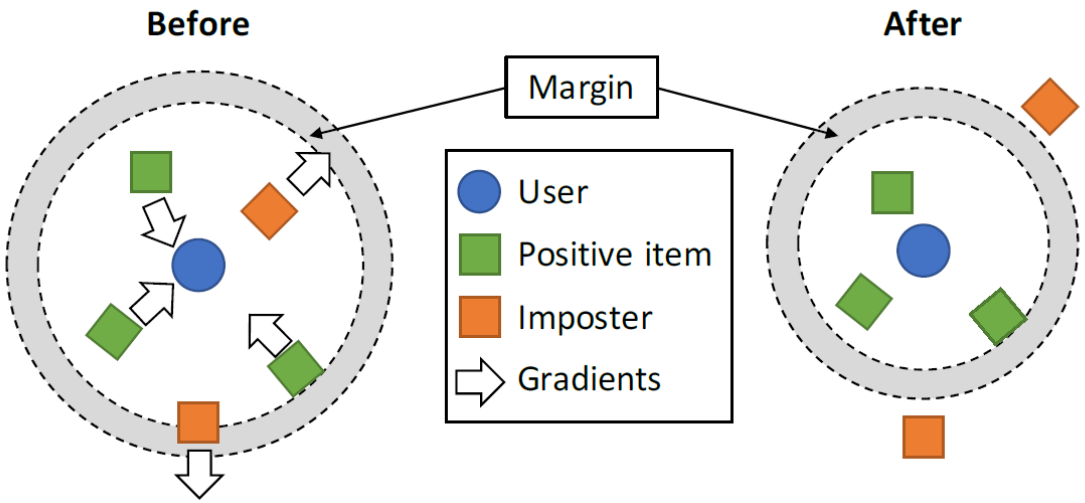
i: user
J: the item user I liked
K: the item user don't like
[]+ = max(z,0) standard hinge loss
W = raning loss weight
M: margine size

Model description

Embedding Loss, triplet loss

$$[m + d(i,j)^2 - d(i,k)^2]_+,$$

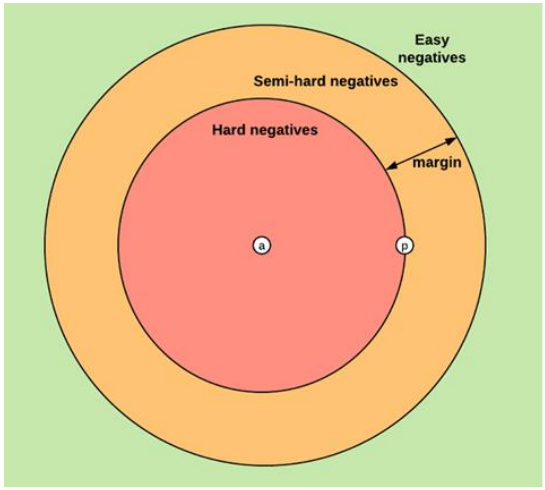




Model description

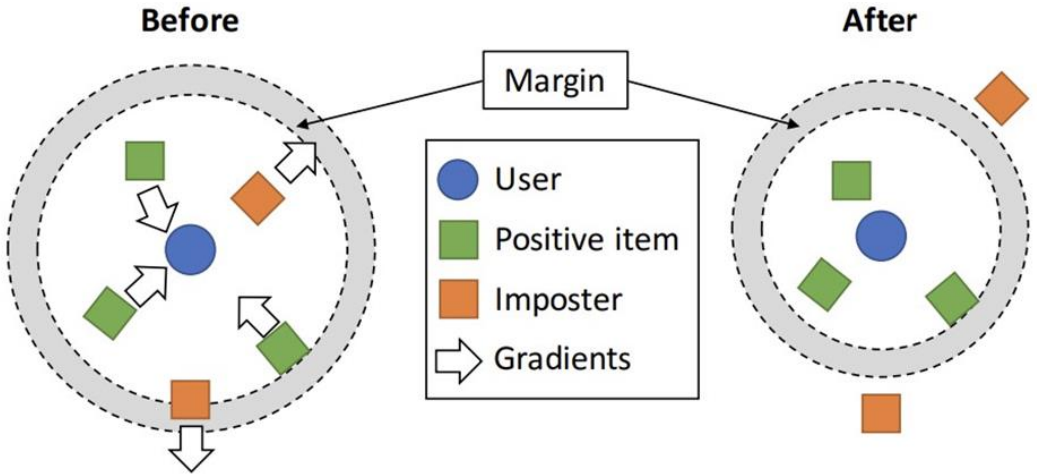
Embedding Loss, triplet loss

$$[m + d(i,j)^2 - d(i,k)^2]_+,$$



1. Easy Triplets

$$d(i,k)^2 > m + d(i,j)^2 \rightarrow \mathcal{L}_m = 0$$



2. Hard Triplets & Semi-Hard Triplets

$$d(i,k)^2 < m + d(i,j)^2$$

Hard Triplet, semi-hard인 경우를 줄여야한다 -> minmize poblem

Model description

Negative Sample mining

$$[m + d(i,j)^2 - d(i,k)^2]_+,$$

Hard Triplet, semi-hard인 case를 위주로 sampling해야 한다고 Facenet 논문에서는 주장

-> CML에는 별다른 얘기는 없음

FaceNet: A Unified Embedding for Face Recognition and Clustering

Florian Schroff
fschroff@google.com
Google Inc.

Dmitry Kalenichenko
dkalenichenko@google.com
Google Inc.

James Philbin
jphilbin@google.com
Google Inc.

Abstract

Despite significant recent advances in the field of face recognition [10, 14, 15, 17], implementing face verification and recognition efficiently at scale presents serious challenges to current approaches. In this paper we present a system, called FaceNet, that directly learns a mapping from face images to a compact Euclidean space where distances directly correspond to a measure of face similarity. Once this space has been produced, tasks such as face recognition, verification and clustering can be easily implemented using standard techniques with FaceNet embeddings as feature vectors.

Our method uses a deep convolutional network trained to directly optimize the embedding itself, rather than an intermediate bottleneck layer as in previous deep learning approaches. To train, we use triplets of roughly aligned matching / non-matching face patches generated using a novel online triplet mining method. The benefit of our approach is much greater representational efficiency: we achieve state-of-the-art face recognition performance using only 128-bytes per face.

On the widely used Labeled Faces in the Wild (LFW) dataset, our system achieves a new record accuracy of ***99.63%****. On YouTube Faces DB it achieves* ***95.12%****. Our system cuts the error rate in comparison to the best published result [15] by 30% on both datasets.*

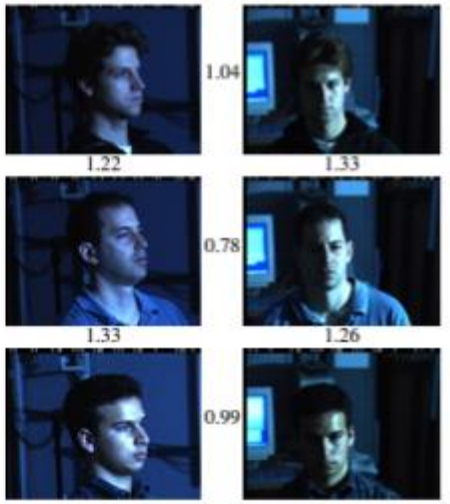


Figure 1. **Illumination and Pose invariance.** Pose and illumination have been a long standing problem in face recognition. This figure shows the output distances of FaceNet between pairs of faces of the same and a different person in different pose and illumination combinations. A distance of 0.0 means the faces are identical, 4.0 corresponds to the opposite spectrum, two different identities. You can see that a threshold of 1.1 would classify every

Model description

Embedding Loss(rank-based weighting scheme)

$$w_{ij} = log(rank_d(i, j) + 1).$$

Rank에 따른 penalize system: rank가 낮다 = 0에 가깝다 =
user i가 item j를 선호하는 순위

User i에 대해서 Positive Item j가 User i에 대해서 가장 가까이 있지 않을 때 패널티를 부과하여 User i에 가까워지도록 설정해 준 가중치

*wij*는 순위를 매겨야 하기 때문에 User와 Positive 각각 한 쌍, 그리고 여러 Negative samples를 필요로 하여 시간 비용이 많이 소모 -> 단순화가 필요하다

1.반복문으로 Hard Triplets & Semi-Hard Triplets에 있는 Negative를 찾는다.
2.찾은 Negative samples의 거리를 계산한 뒤 해당 (User, Positive)에 대해 순위를 매기고 가중치를 계산한다.

Model description

Embedding Loss(rank-based weighting scheme)

$$w_{ij} = log(rank_d(i,j) + 1).$$

근사 방법 1: $rank_d(i,j)$ is then approximated as $\lfloor \frac{J}{N} \rfloor$.

J: 전체 item의 개수
N: 하나의 imposter을 찾을 때까지 반복한 횟수

각각의 item들이 뽑힐 확률이 1/J이기에 기하분포에 따라 평균에 근사하면 식이 유도됨

이 또한 N의 크기가 커질 것을 우려하여 실험에서는 U를 10 or 20으로 둠(실제 test에서 이 값을 조절함)

Model description

Embedding Loss(rank-based weighting scheme)

$$w_{ij} = log(rank_d(i, j) + 1).$$

최종 근사 방법: $rank_d(i, j)$ is then approximated as $\lfloor \frac{J \times M}{U} \rfloor$

J: 전체 item의 개수
N: 하나의 imposter을 찾을 때까지 반복한 횟수
U: 10~20
M: U sample에서 imposter의 개수

Model description

Integrating Item features(Feature loss)

$$\mathcal{L}_f(\theta, v_*) = \sum_j \|f(x_j, \theta) - v_j\|^2$$

$f(x)$: MLP (capture item j's charaterristics)

x_j : Item j의 feature (text description, tag, image pixel)

v_j : Item j의 vector

*Lf*를 통해 Item feature가 유사한 Item들끼리 서로 클러스터링 되는 결과를 가지게 된다.

Model description

Regularization(Covariance loss)

$$\min_{u_*, v_*} \sum_{r_{ij} \in \mathcal{K}} (r_{ij} - u_i^T v_j)^2 + \lambda_u \|u_i\|^2 + \lambda_v \|v_i\|^2)$$

기존 matrix factorization은 위와 같은 regularization term 이용
-> CML은 distance 기반임으로 해당 식과 같이 regularization 식 정의하면 비효율적
(origin in our metric space does not have any specic meaning.)

$$\|\mathbf{u}_*\|^2 \leq 1 \text{ and } \|\mathbf{v}_*\|^2 \leq 1,$$

Model description

Regularization(Covariance loss)

$$C_{ij} = \frac{1}{N}\sum_n (y_i^n - \mu_i)(y_j^n - \mu_j),$$

$$C = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,m} \end{bmatrix}$$

where $\mu_i = \frac{1}{N}\sum_n y_i^n$. We define the loss $\mathcal{L}_c$ to regu
the covariances:

$$\mathcal{L}_c = \frac{1}{N}(\|C\|_f - \|diag(C)\|_2^2),$$

$$\|C\|_f^2 = |a_{11}|^2 + |a_{12}|^2 + \cdots + |a_{mm}|^2$$

$$\|diag(C)\|_2^2 = |a_{11}|^2 + |a_{22}|^2 + |a_{33}|^2 + \cdots + |a_{mm}|^2$$

공분산을 minimize하기 위한 것이기에 분산부분은 제거

Model description

Regularization(Covariance loss)

$$C_{ij} = \frac{1}{N}\sum_n (y_i^n - \mu_i)(y_j^n - \mu_j),$$

$$C = \begin{bmatrix} a_{1,1} & \cdots & a_{1,m} \\ \vdots & \ddots & \vdots \\ a_{m,1} & \cdots & a_{m,m} \end{bmatrix}$$

where $\mu_i = \frac{1}{N}\sum_n y_i^n$. We define the loss $\mathcal{L}_c$ to regu
the covariances:

$$\mathcal{L}_c = \frac{1}{N}(\|C\|_f - \|diag(C)\|_2^2),$$

$$\|C\|_f^2 = |a_{11}|^2 + |a_{12}|^2 + \cdots + |a_{mm}|^2$$

$$\|diag(C)\|_2^2 = |a_{11}|^2 + |a_{22}|^2 + |a_{33}|^2 + \cdots + |a_{mm}|^2$$

공분산을 minimize하기 위한 것이기에 분산부분은 제거

Model description

Final model

1. Sample N positive pairs from $\mathcal{S}$
2. For each pair, sample U negative items and approximate $rank_d(i, j)$ as described in Section 3.2
3. For each pair, keep the negative item k that maximizes the hinge loss and form a mini-batch of size N.
4. Compute gradients and update parameters with a learning rate controlled by AdaGrad.
5. Censor the norm of $\mathbf{u}_*$ and $\mathbf{v}_*$ by $\mathbf{y}' = \frac{\mathbf{y}}{\max(\|\mathbf{y}\|, 1)}$.
6. Repeat this procedure until convergence.

$$\min_{\theta, \mathbf{u}_*, \mathbf{v}_*} \mathcal{L}_m + \lambda_f \mathcal{L}_f + \lambda_c \mathcal{L}_c$$

$$\text{s.t.} \quad \|\mathbf{u}_*\|^2 \leq 1 \text{ and } \|\mathbf{v}_*\|^2 \leq 1,$$

We minimize this constrained objective function with Mini-Batch Stochastic Gradient Descent (SGD) and control the learning rating using AdaGrad [10], as suggested in

•AdaGrad does not seem to work on GPU. Try using AdamOptimizer instead

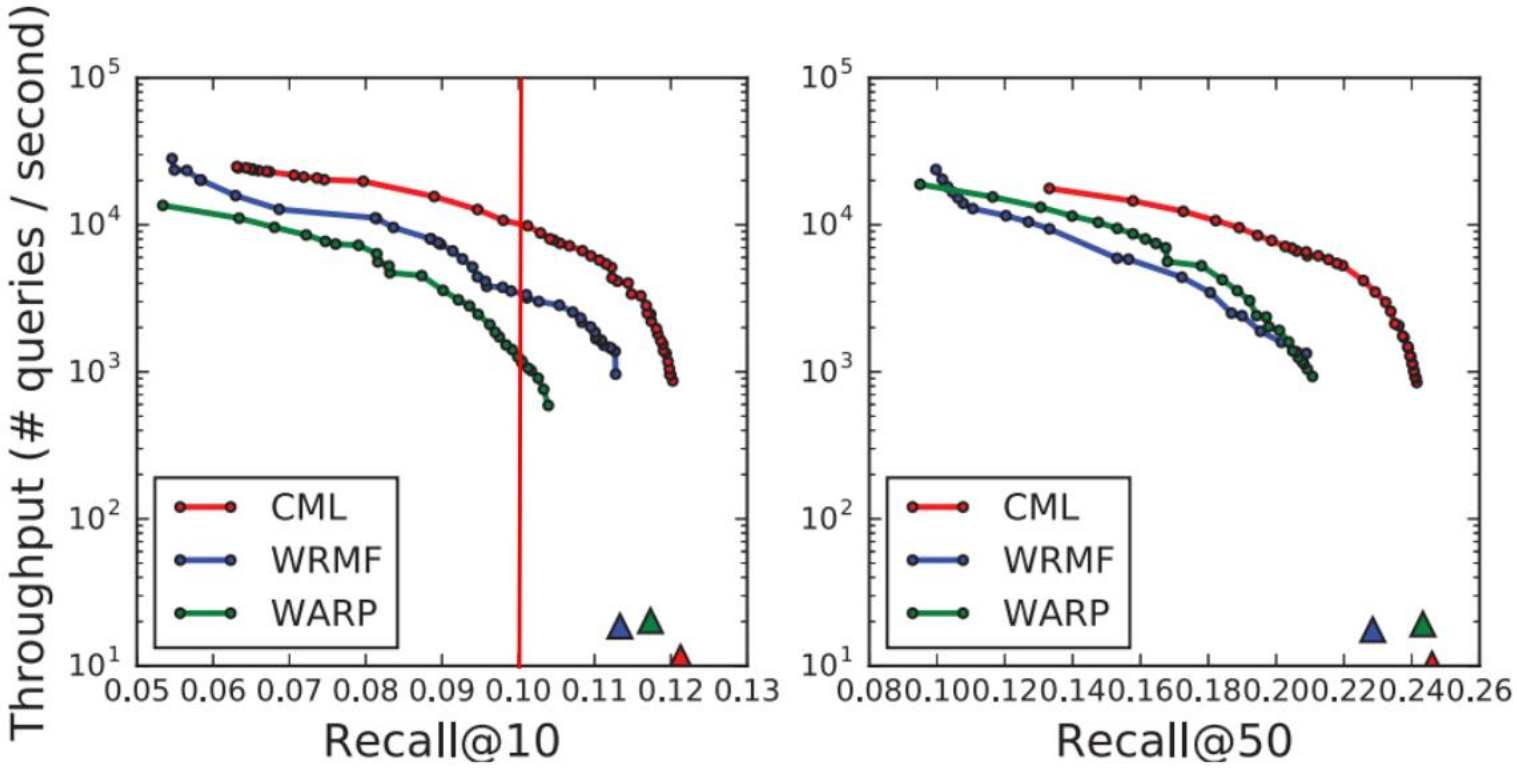
Experiment

Dataset

	WRMF	BPR	WARP	CML	*ours vs. best*	FM	VBPR	CDL	CML+F	*ours vs. best*
					Recall@50					
CiteULike	0.2437	0.2489	0.1916	**0.2714*** **	9.03%	0.1668	0.2807	**0.3375** **	0.3312	-1.86%
BookCX	0.0910	0.0812	0.0801	**0.1037*** **	13.95%	0.1016	0.1004	0.0984	**0.1147*** **	12.89%
Flickr	0.0667	0.0496	0.0576	**0.0711*** **	6.59%	NA	0.0612	0.0679	**0.0753*** **	10.89%
Medium	0.1457	0.1407	0.1619	**0.1730*** **	6.41%	0.1298	0.1656	0.1682	**0.1780*** **	5.82%
MovieLens	0.4317	0.3236	0.4649	**0.4665**	0.34%	0.4384	0.4521	0.4573	**0.4617** *	0.96%
EchoNest	0.2285	0.1246	0.2433	**0.2460**	1.10%	NA	NA	NA	NA	NA
					Recall@100					
CiteULike	0.3112	0.3296	0.2526	**0.3411*** **	3.37%	0.2166	0.3437	0.4173	**0.4255** **	1.96%
BookCX	0.1286	0.1230	0.1227	**0.1436*** **	11.66%	0.1440	0.1455	0.1428	**0.1712*** **	17.66%
Flickr	0.0821	0.0790	0.0797	**0.0922*** **	12.30%	NA	0.0880	0.0909	**0.1048*** **	15.29%
Medium	0.2112	0.2078	0.2336	**0.2480*** **	6.16%	0.1900	0.2349	0.2408	**0.2531*** **	5.10%
MovieLens	0.5649	0.4455	0.5989	**0.6022**	0.55%	0.5561	0.5712	0.5943	**0.5976**	0.55%
EchoNest	0.2891	0.1655	0.3021	**0.3022**	0.00%	NA	NA	NA	NA	NA

$$Recall@k = \frac{\sharp\ of\ recommended\ items\ @k\ that\ are\ relevant}{total\ \sharp\ of\ relevant\ items}$$

Experiment

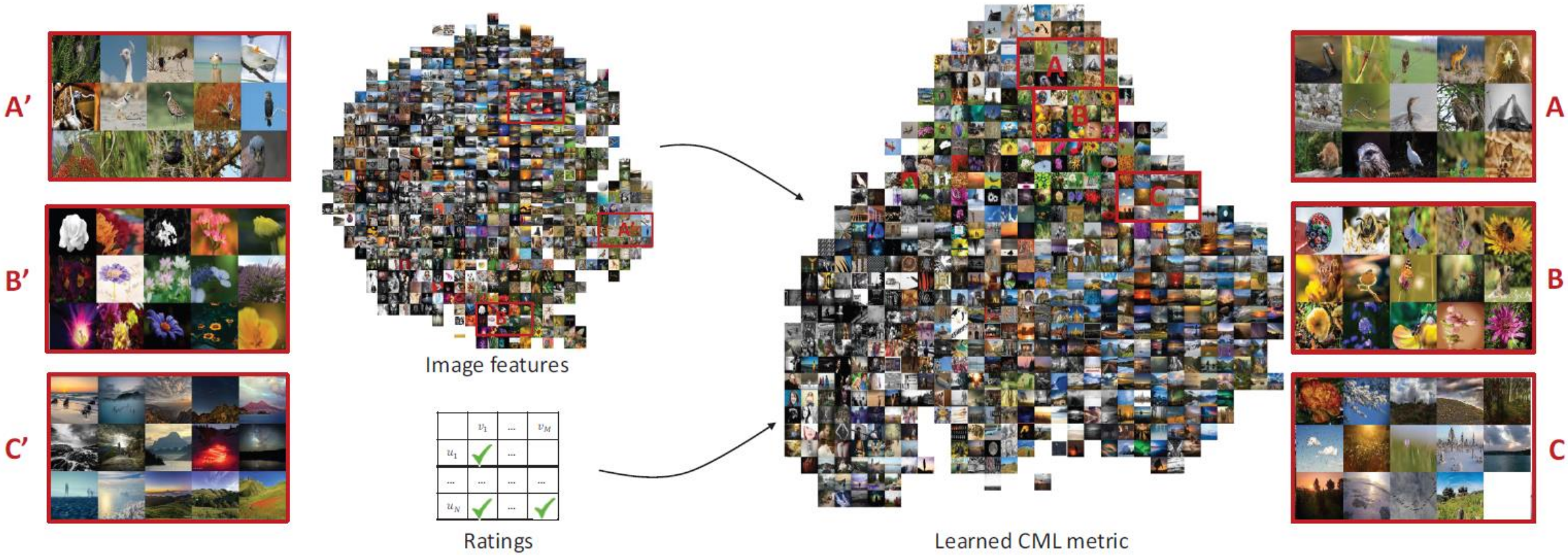


1. CML gains 106x and 86x speedup with only 2% of reduction in recall@10 and recall@50.

2. fastest among the three algorithms given the same accuracy. For example, given recall@10= 0:1, CML is more than 8x faster than other MF algorithms.

Experiment

Dataset



Model Implementation

Dataset&Dataloader

- User dataset: Movie lens
- If rating is same or over than 4, take is as positive(1)
- Ow 0
- Use data that have more than 10 rates for accuracy
- Test at racall@50
- Train 0.6, Validation data 0.2, test 0.2

Model Implementation

Dataset&Dataloader

```python
class MovieLensDataLoader(BaseDataLoader):
    """
    Movielens data loading demo using BaseDataLoader
    """
    def __init__(self, data_dir, batch_size, shuffle=True, validation_split=0.0, num_workers=1, training=True):
        self.data_dir = data_dir
        self.dataset = MovieLensDataset(os.path.join(data_dir, 'ml-100k/u.data'),20)
        super().__init__(self.dataset, batch_size, shuffle, validation_split, num_workers)
```

MovieLens data [13], we include the ratings
greater or equal to 4 as positive feedback as suggested
by
the prior works on implicit feedback [38, 47], and
include
the users with more than 10 ratings. p6

```python
class MovieLensDataset(Dataset):
    """
    MovieLens dataset loading for CML
    """
    def __init__(self, data_file, neg_sample_size):
        self.data = pd.read_csv(data_file, sep='\t', header=None)
        self.data[2] = self.data[2].apply(self.__preprocess_rating) #preprocess rating change
        self.data = self.data.groupby(0).filter(lambda x: (x[2] == 1).sum() >= 10)
        # Filter rows, get data with more than 10 ratings
        self.n_users = self.data[0].nunique()
        self.n_items = self.data[1].nunique()
        self.neg_sample_size = neg_sample_size
        self.users = self.data[0].unique()
        self.posdata = self.data[self.data[2] == 1].groupby(0)[1].apply(list).to_dict()
        self.negdata = self.data[self.data[2] == 0].groupby(0)[1].apply(list).to_dict()

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        user = self.users[idx]
        item = self.posdata[user]
        neg_item = np.random.choice(self.negdata[user], self.neg_sample_size).tolist()
        return {'user':user,
                'pos_item':item,
                'neg_item':neg_item,
                }

    #change rate to implicit data
    def __preprocess_rating(self, rating):
        if rating >= 4:
            return 1
        else:
            return 0
```

Model Implementation

Model

```python
def _feature_projection(self):
    """
    :return: the projection of the feature vectors to the user-item embedding
    """
    # feature loss
    if self.features is not None:
        # fully-connected layer
        output = self.mlp(self.features) * self.feature_projection_scaling_factor
        # projection to the embedding
        return clip_by_norm(output, self.clip_norm)

def _feature_loss(self):
    """
    :return: the l2 loss of the distance between items' their embedding and their feature projection
    """
    loss = torch.tensor(0, dtype=torch.float32)
    feature_projection = self._feature_projection()
    if feature_projection is not None:
        # apply regularization weight
        loss += torch.sum((self.item_embeddings.weight.data - feature_projection) ** 2) * self.feature_l2_reg
    return loss
```

Model Implementation

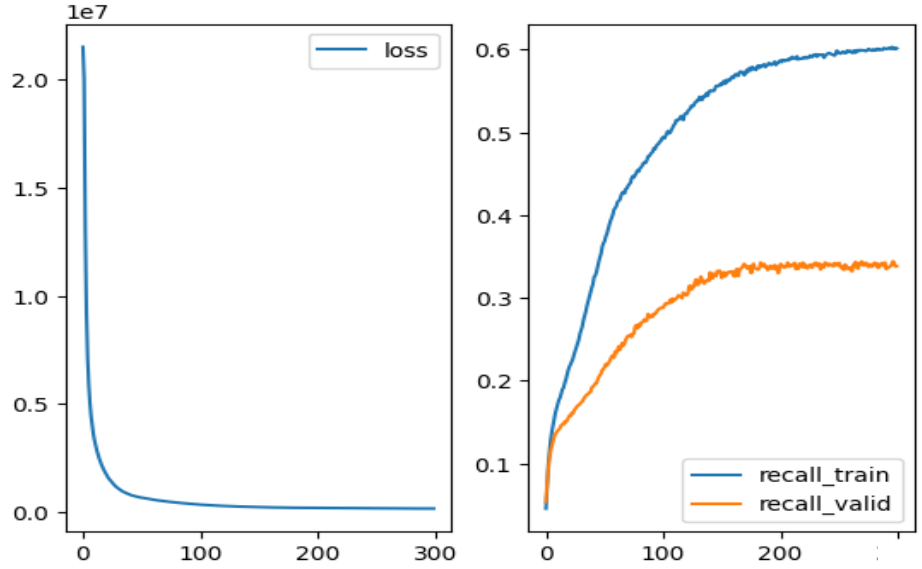
```python
def _embedding_loss(self, X):
    """
    :return: the distance metric loss
    """
    # Let
    # N = batch size,
    # K = embedding size,
    # W = number of negative samples per a user-positive-item pair
    # user embedding (N, K)
    users = self.user_embeddings(X[:, 0])

    # positive item embedding (N, K)
    pos_items = self.item_embeddings(X[:, 1])
    # positive item to user distance (N)
    pos_distances = torch.sum((users - pos_items) ** 2, 1)

    # negative item embedding (N, K, W)
    neg_items = self.item_embeddings(X[:, 2:]).transpose(-2, -1)
    # distance to negative items (N x W)
    distance_to_neg_items = torch.sum((users.unsqueeze(-1) - neg_items) ** 2, 1)

    # best negative item (among W negative samples) their distance to the user embedding (N)
    closest_negative_item_distances = distance_to_neg_items.min(1)[0]

    # compute hinge loss (N)
    distance = pos_distances - closest_negative_item_distances + self.margin
    loss_per_pair = F.relu(distance)

    if self.use_rank_weight:
        # indicator matrix for impostors (N x W)
        impostors = (pos_distances.unsqueeze(-1) - distance_to_neg_items + self.margin) > 0
        # approximate the rank of positive item by (number of impostor / W per user-positive pair)
        rank = impostors.float().mean(1) * self.n_items
        # apply rank weight
        loss_per_pair *= torch.log(rank + 1)

    # the embedding loss
    loss = loss_per_pair.sum()
    return loss
```

```python
def _covariance_loss(self):
    X = torch.cat((self.item_embeddings.weight.data, self.user_embeddings.weight.data), 0)
    n_rows = X.shape[0]
    X -= X.mean(0)
    cov = torch.matmul(X.transpose(-2, -1), X) / n_rows
    loss = cov.sum() - cov.trace()
    return loss * self.cov_loss_weight
```

```python
def loss(self, X):
    """
    :return: the total loss = embedding loss + feature loss
    """
    X = X.to(self.device)
    feature_loss = self._feature_loss()
    loss = self._embedding_loss(X) + feature_loss
    if self.use_cov_loss:
        loss += self._covariance_loss()
    return loss, feature_loss
```

Model Implementation

Model




```
train recall@50 0.6016, test recall@50
0.3384
```

	WRMF	BPR	WARP	CML	*ours vs. best*	FM	VBPR	CDL	CML+F	*ours vs. best*
					Recall@50					
CiteULike	0.2437	0.2489	0.1916	**0.2714*****	9.03%	0.1668	0.2807	**0.3375****	0.3312	-1.86%
BookCX	0.0910	0.0812	0.0801	**0.1037*****	13.95%	0.1016	0.1004	0.0984	**0.1147*****	12.89%
Flickr	0.0667	0.0496	0.0576	**0.0711*****	6.59%	NA	0.0612	0.0679	**0.0753*****	10.89%
Medium	0.1457	0.1407	0.1619	**0.1730*****	6.41%	0.1298	0.1656	0.1682	**0.1780*****	5.82%
MovieLens	0.4317	0.3236	0.4649	**0.4665**	0.34%	0.4384	0.4521	0.4573	**0.4617***	0.96%
EchoNest	0.2285	0.1246	0.2433	**0.2460**	1.10%	NA	NA	NA	NA	NA
					Recall@100					
CiteULike	0.3112	0.3296	0.2526	**0.3411*****	3.37%	0.2166	0.3437	0.4173	**0.4255****	1.96%
BookCX	0.1286	0.1230	0.1227	**0.1436*****	11.66%	0.1440	0.1455	0.1428	**0.1712*****	17.66%
Flickr	0.0821	0.0790	0.0797	**0.0922*****	12.30%	NA	0.0880	0.0909	**0.1048*****	15.29%
Medium	0.2112	0.2078	0.2336	**0.2480*****	6.16%	0.1900	0.2349	0.2408	**0.2531*****	5.10%
MovieLens	0.5649	0.4455	0.5989	**0.6022**	0.55%	0.5561	0.5712	0.5943	**0.5976**	0.55%
EchoNest	0.2891	0.1655	0.3021	**0.3022**	0.00%	NA	NA	NA	NA	NA