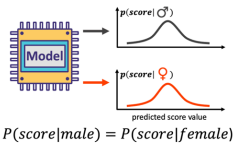
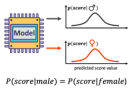
Fairness definition: score-based statistical parity



$$P(score|male) = P(score|female)$$

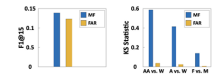
Statistical parity requires the **prediction** (predicted scores) from the model to be **independent** to the **group labels**.

Takeaways



- Define recommendation fairness as fairness. **score-based statistical parity**.

- Propose an **embedding intervention** method to enhance

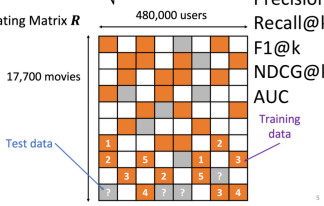


- Effective performance for the challenging **multi-attribute** and **multi-category** scenario.

So far, focus on explicit recommendation

$$RMSE = \sqrt{\frac{1}{n}\sum_{u,i}(\hat{r}_{u,i} - r_{u,i})^2}$$

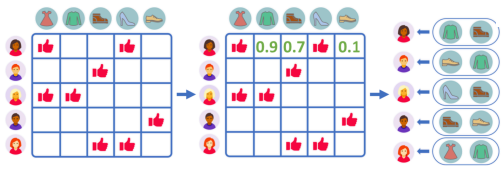
Rating Matrix $\boldsymbol{R}$



Precision@k
Recall@k
F1@k
NDCG@k
AUC

Implicit Recommendation

A user set $\mathcal{U} = \{1,2,\dots,N\}$; an item set $\mathcal{I} = \{1,2,\dots,M\}$; a binary variable $y_{u,i}$; an implicit feedback dataset $\mathcal{O} = \{(u,i)\}$; and the goal is to predict $\hat{y}_{u,i}$ when $y_{u,i} = 0$, and return a ranked list of k items based on $\hat{y}_{u,i}$.



$$Precision@k = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}\frac{|\mathcal{O}_u^{test}\cap\mathcal{O}_u^{rec}|}{k}$$

$$Recall@k = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}\frac{|\mathcal{O}_u^{test}\cap\mathcal{O}_u^{rec}|}{|\mathcal{O}_u^{test}|}$$

$$F1@k = \frac{2*precision@k*recall@k}{precision@k+recall@k}$$

Do not consider different importance of different ranking positions

$$AUC(u) = \frac{1}{|\mathcal{O}_u^{test}||\mathcal{I}\setminus\mathcal{O}_u^{test}|}\sum_{i\in\mathcal{O}_u^{test}}\sum_{j\in\mathcal{I}\setminus\mathcal{O}_u^{test}}\mathbb{I}(\hat{y}_{u,i} > \hat{y}_{u,j})$$

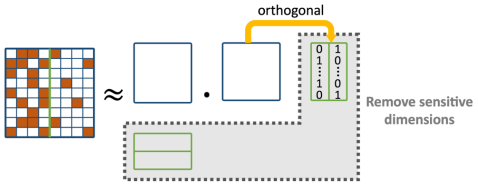
Area Under the Curve

Probability of a positive item being ranked higher than a negative item.

$$\mathcal{L}_{BPR} = -\sum_{(u,i,j)\in\mathcal{O}^{train}}\log(\sigma(\boldsymbol{P}_u^T\boldsymbol{Q}_i - \boldsymbol{P}_u^T\boldsymbol{Q}_j))$$

- Gradient Descent to solve the minimization problem
- Add regularization to counteract overfitting

Fairness-Aware Recommendation (FAR) algorithm



Global method – Data Augmentation algorithm (DA)

- For a niche user u, generate synthetic users based on the distribution (following the **Distribution Calibration** method[1]):

Historical interaction record of u.

$$\mathbf{p}_u = \alpha\mathbf{O}_u + (1-\alpha)\frac{1}{|\mathcal{N}_u|}\sum_{v\in\mathcal{N}_u}\mathbf{O}_v$$

Neighbor users similar to u.

Data from users of different preferences may **not be helpful (even play negative roles)** for learning preference of a target group of users.

Global methods have to **trade-off** between niche and mainstream users.

	ML1M					
	NDCG @20	Subgroups of mainstream levels				
		low	med-low	medium	med-high	high
VAE	0.315	0.209	0.264	0.283	0.337	0.483
DA	0.317	0.222	0.266	0.282	0.335	0.480
WL	0.319	0.232	0.272	0.284	0.332	0.476
LOCA	0.323	0.242	0.276	0.286	0.335	0.476
LFT	**0.337**	**0.255**	**0.288**	**0.298**	**0.349**	**0.496**
LFT vs. VAE	6.98%	22.01%	9.09%	5.30%	3.54%	2.69%
LFT vs. LOCA	4.40%	5.55%	4.09%	4.23%	4.24%	4.22%

Proposed LFT outperforms other alternatives significantly.

Similarity-based approach

Mainstream users have high similarity to other users.

Jaccard similarity between u and v w.r.t. historical interactions

Number of users

$$MS_u^{sim} = \sum_{v\in\mathcal{U}\setminus u} J_{u,v}/(N-1).$$

Average similarity to other users

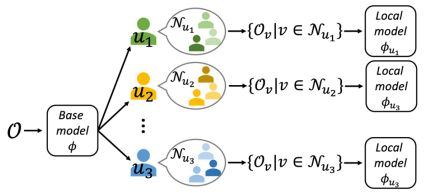
Global method – Weighted Loss algorithm (WL)

- Add **weights** to users in loss function.
- Users with lower mainstream levels get higher weights:

$$\mathcal{L}_{WL} = \sum_{u\in\mathcal{U}} w_u \cdot \mathcal{L}_{VAE}(u), \quad w_u \propto (\frac{1}{MS_u})^\beta$$

Original loss for user u

Local method – Local Fine Tuning algorithm



Treat recommending for each user as an independent task using a **customized local model**.

How?

- $DCG@k = \sum_{i=1}^{k}\mathbb{I}(y_{u,i}=1)\frac{1}{\log(1+i)}$

Discounted Cumulative Gain

- $IDCGk = \sum_{i=1}^{\min(|\mathcal{O}_u^{test}|,k)}\frac{1}{\log(1+i)}$

Ideal Discounted Cumulative Gain

- $NDCG@k = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}\frac{DCG@k}{IDCG@k}$

Normalized Discounted Cumulative Ga

$$AUC = \frac{1}{|\mathcal{U}|}\sum_{u\in\mathcal{U}}AUC(u)$$

User-user CF

$$\hat{r}_{u,i} = \bar{r}_u + \frac{\sum_{u'\in\mathcal{N}}s(u,u')(r_{u',i}-\bar{r}_{u'})}{\sum_{u'\in\mathcal{N}}s(u,u')}$$

RMSE loss can be replaced by **cross-entropy loss**

$$\boldsymbol{P},\boldsymbol{Q} = \arg\min_{\boldsymbol{P},\boldsymbol{Q}}\mathcal{L}$$

$$= -\sum_{(u,i)\in\mathcal{O}^{train}}(y_{u,i}\log(\sigma(\boldsymbol{P}_u^T\boldsymbol{Q}_i)) + (1-y_{u,i})\log(1-\sigma(\boldsymbol{P}_u^T\boldsymbol{Q}_i))) + \lambda(\|\boldsymbol{P}\|+\|\boldsymbol{Q}\|)$$

Item-item CF

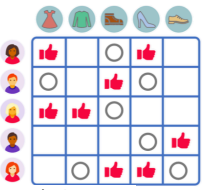
$$\hat{y}_{u,i} = \frac{\sum_{i'\in\mathcal{N}}s(i,i')y_{u,i'}}{\sum_{i'\in\mathcal{N}}s(i,i')}$$

Matrix Factorization

$$\hat{y}_{u,i} = \frac{\sum_{u'\in\mathcal{N}}s(u,u')y_{u',i}}{\sum_{u'\in\mathcal{N}}s(u,u')}$$

Randomly sample user-item pairs with $y_{u,i} = 0$ as negative samples. (change every epoch)

$$\mathcal{O}^{train} = \mathcal{O}\cup\mathcal{O}^{neg}$$



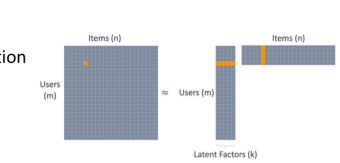
MF for Implicit Recommendation

Matrix Factorization

$$\boldsymbol{P},\boldsymbol{Q} = \arg\min_{\boldsymbol{P},\boldsymbol{Q}}\mathcal{L}$$

$$= \sum_{(u,i)\in\mathcal{O}^{train}}(\boldsymbol{P}_u^T\boldsymbol{Q}_i - y_{u,i})^2 + \lambda(\|\boldsymbol{P}\|+\|\boldsymbol{Q}\|)$$

where $\mathcal{O}^{train} = \mathcal{O}\cup\mathcal{O}^{neg}$



$$\boldsymbol{P},\boldsymbol{Q} = \arg\min_{\boldsymbol{P},\boldsymbol{Q}}\mathcal{L}$$

$$= \sum_{(u,i)\in\mathcal{O}}(\hat{y}_{u,i} - y_{u,i})^2 + regularization$$

$$= \sum_{(u,i)\in\mathcal{O}}(\boldsymbol{P}_u^T\boldsymbol{Q}_i - y_{u,i})^2 + \lambda(\|\boldsymbol{P}\|+\|\boldsymbol{Q}\|)$$

BPR: Bayesian Personalized Ranking

$$AUC(u) = \frac{1}{|\mathcal{O}_u^{test}||\mathcal{I}\setminus\mathcal{O}_u^{test}|}\sum_{i\in\mathcal{O}_u^{test}}\sum_{j\in\mathcal{I}\setminus\mathcal{O}_u^{test}}\mathbb{I}(\hat{y}_{u,i} > \hat{y}_{u,j})$$

Main idea: to maximize AUC by making all positive items ranked higher than negative items.

During training, we do not have access to $\mathcal{O}_u^{test}$, but we have $\mathcal{O}_u$ and $\mathcal{O}_u^{neg}$.

Micro Ave. Table

		Truth: no	Truth: yes
Classifier: yes		20	100
Classifier: no		1860	20

Class 2

		Truth: no	Truth: yes
Classifier: yes		10	90
Classifier: no		890	10

Class 1

		Truth: no	Truth: yes
Classifier: yes		10	10
Classifier: no		970	10

Macroaveraged precision: (0.5 + 0.9)/2 = 0.7

Macroaveraged precision: 100/120 = .83

Microaveraged score is dominated by score on frequent classes

```
miu = train_df["Rating"].mean()
unique_users_in_training = train_df["UserID"].unique()
unique_movies_in_training = train_df["MovieID"].unique()

mean_user_ratings = {}
for each in unique_users_in_training:
    looped_df = train_df.loc[train_df["UserID"] == each]
    mean_user_ratings[each] = looped_df["Rating"].mean()

mean_item_rating = {}
for each in unique_movies_in_training:
    looped_df = train_df.loc[train_df["MovieID"] == each]
    mean_item_rating[each] = looped_df["Rating"].mean()

pred_mat = []

for user_index in range(0, UNIQUE_USERS):
    movie_matrix = []
    for movie_index in range(0, UNIQUE_MOVIES):
        if ((movie_index not in unique_movies_in_training) or (mean_item_rating[movie_index] == 0.0)
            b_i = 0
        else:
            b_i = mean_item_rating[movie_index] - miu

        if ((user_index not in unique_users_in_training) or (mean_user_ratings[user_index] == 0.0))
            b_u = 0
        else:
            b_u = mean_user_ratings[user_index] - miu

        movie_matrix.append(miu + b_i + b_u)

    pred_mat.append(movie_matrix)
```

BPR Loss to Optimize AUC

$$AUC(u) = \frac{1}{|O_u||O_u^{neg}|} \sum_{i \in O_u} \sum_{i \in O_u^{neg}} \mathbb{I}(\hat{y}_{u,i} > \hat{y}_{u,j})$$

$$\mathcal{L}_{BPR}(u) = -\sum_{i \in O_u} \sum_{i \in O_u^{neg}} \log(\sigma(\hat{y}_{u,i} - \hat{y}_{u,j}))$$

$$\mathcal{L}_{BPR} = -\sum_{(u,i,j) \in O^{train}} \log(\sigma(\hat{y}_{u,i} - \hat{y}_{u,j}))$$

```
## Shuffle the dataset
ds = list(zip(self.sample_user, self.sample_movie))
np_rng.shuffle(ds)

batch_gradient_updates = False

## Batching updates Part 1 of 3
if batch_gradient_updates:
    (gLp_updates, gLq_updates) = ({}, {})


for u, i in ds:
    if self.train_indicator_mat[u, i] == 0.0:
        continue
    r_ui = self.train_mat[u, i]
    (Pu, Qi) = (self.P[u], self.Q[i])
    ls = 2 * (Pu @ Qi - r_ui)
    gL_Pu = ((ls * Qi) + (self.reg * Pu))
    gL_Qi = ((ls * Pu) + (self.reg * Qi))

    self.P[u] -= self.lr * gL_Pu
    self.Q[i] -= self.lr * gL_Qi

def masked_rmse(x,y, mask=1):
    inner = (((x*mask) - (y*mask)) ** 2)
    o = (np.sum( inner) / np.sum(mask))
    return np.sqrt(o)

pred = np.matmul(self.P, self.Q.T)
regularization = (self.reg * (np.sum(self.P**2) + np.sum(self.Q**2)))
epoch_loss  = (np.sum((pred*self.train_indicator_mat - self.train_mat)**2)) + regularization
```

```
def get_support(arr, *items):
    ## Returns the mean support of an iterable containing items that are valid indices of our ohe array
    # Equivalent to ( summation_N[summation_K[(user0_item0 ... U ... userK_itemN)] / K)] / N,
    # where K is the number of users and N is the number of items in the iterable
    return np.product([arr[:,item] for item in items], axis=0).mean()
```

completeness:

$$c = 1 - \frac{H(K|C)}{H(K)}$$

Probability of a sample in class c is in cluster k

$$H(K|C) = -\sum_{c=1}^{C} \sum_{k=1}^{K} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{k=1}^{K} a_{ck}}$$

$$H(K) = -\sum_{k=1}^{K} \frac{\sum_{c=1}^{C} a_{ck}}{N} \log \frac{\sum_{c=1}^{C} a_{ck}}{N}$$

Entropy of the whole dataset w.r.t cluster labels

eneity:

$$h = 1 - \frac{H(C|K)}{H(C)}$$

Probability of a sample in cluster k is class c

$$H(C|K) = -\sum_{k=1}^{K} \sum_{c=1}^{C} \frac{a_{ck}}{N} \log \frac{a_{ck}}{\sum_{c=1}^{C} a_{ck}}$$

$$H(C) = -\sum_{c=1}^{C} \frac{\sum_{k=1}^{K} a_{ck}}{N} \log \frac{\sum_{k=1}^{K} a_{ck}}{N}$$

Entropy of the whole dataset w.r.t class labels

Rule Evaluation Metrics

- Support (s)
  - Fraction of transactions that contain both X and Y
- Confidence (c)
  - How often Y appears in transactions that contain X
  - conf(X→Y)=P(Y|X)=$\frac{\sigma(X \cup Y)}{\sigma(Y)}$

TID	Items
1	Bread, Milk
2	Bread, Diaper, Beer, Eggs
3	Milk, Diaper, Beer, Coke
4	Bread, Milk, Diaper, Beer
5	Bread, Milk, Diaper, Coke

Example:

$$\{\text{Milk, Diaper}\} \Rightarrow \text{Beer}$$

$$s = \frac{\sigma(\text{Milk, Diaper, Beer})}{|\text{T}|} = \frac{2}{5} = 0.4$$

$$c = \frac{\sigma(\text{Milk, Diaper, Beer})}{\sigma(\text{Milk, Diaper})} = \frac{2}{3} = 0.67$$

We will change $y_n \in \mathbb{R}$ to a C-dimensional vector (one hot encoding)

$$\mathbf{y_n} = [y_{n1}, \ldots, y_{nC}]^T \in \mathbb{R}^C$$

$$y_{nc} = \begin{cases} 1, & \text{if } y_n = c \\ 0, & \text{otherwise} \end{cases}$$

e.g. if $y_n = 3$ then $\mathbf{y_n} = [0, 0, 1, 0, \ldots, 0]^T \in \mathbb{R}^C$

We will maximize the likelihood

$$L(\mathcal{D}|\mathbf{w_1}, \ldots, \mathbf{w_C}) = \prod_{n=1}^{N} p(\mathbf{y_n}|\mathbf{x_n})$$

$$= \prod_{n=1}^{N} (p(y_{n1} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})^{y_{n1}} \ldots p(y_{nC} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})^{y_{nC}})$$

Data-likelihood

$$L(\mathcal{D}|\mathbf{w_1}, \ldots, \mathbf{w_C}) = \prod_{n=1}^{N} p(y_n|\mathbf{x_n})$$

$$= \prod_{n=1}^{N} (p(y_{n1} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})^{y_{n1}} \ldots p(y_{nC} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})^{y_{nC}})$$

$$= \prod_{n=1}^{N} \prod_{c=1}^{C} p(y_{nc} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})^{y_{nc}}$$

Cross-entropy error (Negative log-likelihood)

$$\mathcal{E}(\mathbf{w_1}, \ldots, \mathbf{w_C}) = -\sum_{n=1}^{N} \sum_{c=1}^{C} y_{nc} \log p(y_{nc} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})$$

Maximum Likelihood Estimation (MLE)

If we don't know the parameter θ of the biased coin but we observe $y_1, .., y_n$ from n independent experiments, then we can estimate θ by maximizing the log likelihood:

$$\theta^* = \arg\max_{\theta} \sum_{i=1}^{n} y_i \log \theta + (1 - y_i) \log(1 - \theta)$$

$$\theta^* = \frac{\sum_{i=1}^{n} y_i}{n}$$

- F_1 or F-score
  - $F = \frac{2 \times P \times R}{P + R} = \frac{2 \cdot 0.9440 \cdot 0.9934}{0.9440 + 0.9934} = 0.9681$
- F_β : weighted combination
  - $F = \frac{(1+\beta^2) \times P \times R}{\beta^2 \times P + R}$

Logistic Regression

- Linear combination of input features $\mathbf{w}^T\mathbf{x}$
- Transform through sigmoid function $\sigma(\mathbf{w}^T\mathbf{x})$ → interpretable as probability
- Decision rule based on whether $\sigma(\mathbf{w}^T\mathbf{x}) \lessgtr 0.5$
- Evaluation through data likelihood, or cross-entropy error

$$\mathcal{E}(\mathbf{w}) = -\sum_{n=1}^{N} \{y_n \log [\sigma(\mathbf{w}^T\mathbf{x_n})] + (1 - y_n) \log [1 - \sigma(\mathbf{w}^T\mathbf{x_n})]\}$$

- Optimization through gradient descent

Multinomial Regression

- Conditional logit model: $p(y = c|\mathbf{x}, \mathbf{w_c}) = \frac{e^{\mathbf{w_c}^T\mathbf{x}}}{\sum_{c=1}^{C} e^{\mathbf{w_c}^T\mathbf{x}}}$
- Similar to 2-class logistic regression
  - compute negative cross-entropy and perform gradient descent

Suppose we have N data samples, C different class labels, K clusters and a_{ck} number of data points belonging to the class c and cluster k.

Homogeneity: $h = 1 - \frac{H(C|K)}{H(C)}$

Completeness: $c = 1 - \frac{H(K|C)}{H(K)}$

V-measure: $V_\beta = \frac{(1 + \beta)hc}{\beta h + c}$

Truth \ Prediction	P	N	Total
P	TP	FN	#P
N	FP	TN	#N
Total	#P̂	#N̂	ALL

Truth \ Prediction	test_Covid = yes	test_Covid = no	Total
is_Covid = yes	6954	46	7000
is_Covid = no	412	2588	3000
Total	7366	2634	10000

Cross-entropy error

$$\mathcal{E}(\mathbf{w_1}, \ldots, \mathbf{w_C}) = -\sum_{n=1}^{N} \sum_{c=1}^{C} y_{nc} \log p(y_{nc} = 1|\mathbf{w_1}, \ldots \mathbf{w_C})$$

- Optimization with gradient descent, convex function
- Computational details are out of scope
- But the gradient vector w.r.t. each weight $\mathbf{w_c}$ looks like this

$$\nabla \mathcal{E}_{\mathbf{w_c}} = \sum_{n=1}^{N} \underbrace{[p(y_{nc} = 1|\mathbf{w_1}, \ldots \mathbf{w_C}) - y_{nc}]}_{\text{error for class c}} \mathbf{x_n}$$

Lift: the ratio between the observed support and the expected support if X and Y were independent

$$Lift(X -> Y) = Lift(Y -> X) = \frac{conf(X -> Y)}{\sup(Y)}$$

$$= \frac{P(Y|X)}{P(Y)} = \frac{P(X,Y)}{P(X)P(Y)} = \frac{\sup(X \cup Y)}{\sup(X)\sup(Y)}$$

- Range [0, ∞)
- Lift(X,Y)=1: X and Y are independent
- Lift(X,Y)>1: X and Y are positively correlated
- Lift(X,Y)<1: X and Y are negatively correlated

```
running_sum = 0.0
count_of_ratings = 0
for user_index in range(0, UNIQUE_USERS):
    movie_matrix = []
    for movie_index in range(0, UNIQUE_MOVIES):
        if test_mat[user_index][movie_index] != 0.0:
            running_sum = running_sum + (test_mat[user_index][movie_index] - pred_mat[user_index][movie_index])**2
            count_of_ratings += 1

RMSE_ERROR = (running_sum/count_of_ratings)**(1/2)
```

```
def kmeansplusplus_centroid_init_fn(x, number_of_centroids):
    centroids = np.zeros((number_of_centroids, x.shape[-1]), dtype=np.float32)
    centroids[0] = x[rng.integers(0, x.shape[0])]
    for i in range(1, number_of_centroids):
        distances = get_distances_to_centroid(x, centroids[i-1])
        probabilities = (distances / np.sum(distances))
        random_indice = rng.choice(x.shape[0], p=probabilities)
        centroids[i] = x[random_indice]
    return centroids
```

```
x = user_movie_data
users = x[:,0]
movies = x[:, 1]

movies_vector_dict = None
users, users_vector_dict = vectorize(users)
movies, movies_vector_dict = vectorize(movies)

x = np.stack([users, movies], axis=1)
arr = one_hot_encode(x)

print(f"  users {users.shape}: \n{users}\n")
print(f"  movies {movies.shape}: \n{movies}\n")
print(f"  np.unique(users).shape: {np.unique(users).shape}")
print(f"  np.unique(movies).shape: {np.unique(movies).shape}")
print(f"  x.shape: {x.shape}")
print(f"  arr.sum(): {arr.sum()}")

## Starter supports
supports = get_supports(arr)
## Quickly prune out our support that isn't needed
supports_mask = np.where(supports >= min_support, 1, 0)
pruned_starter_indices =  np.where(supports >= min_support, )[0]
```

```
def kmeans(x, number_of_centroids:int = 10, maxIt:int = 5, centroid_init_fn=None, debug=False, id=None, labels=None,):

    centroids = centroid_init_fn(x, number_of_centroids)
    lne = None # last mean error

    for i in range(maxIt):

        def get_distances():
            return np.array(list(map(get_distances_to_centroid, zip(rep(x), centroids))))

        points_distances_to_centroids = get_distances()
        minimum_locs = np.argmin(points_distances_to_centroids, axis=0,)


        errors = []
        groups = []
        for j in range(number_of_centroids):
            group_inds = np.where(minimum_locs == j)[0]
            group_members = x[group_inds]
            groups.append(group_members)

            if len(group_members):
                centroids[j] = np.mean(group_members, axis=0)

            group_distances = get_distances_to_centroid(group_members, centroids[j])
            errors.append(np.sum(group_distances**2))

        mean_error = np.sum(errors, )
        (homogeneity, completeness, v_measure) = sklearn.metrics.homogeneity_completeness_v_measure(labels, minimum_locs)

        def prep_return():
            points_distances_to_centroids = pdc = get_distances()
            mloc = np.argmin(pdc, axis=0,)

            (accuracy, preds, _labels) = get_accuracy(mloc, labels)

            rd = dict(
                points=x,
                groups=groups,
                centroids=centroids,
                mean_error=mean_error,
                points_distances_to_centroids=points_distances_to_centroids,
                homogeneity=homogeneity,
                completeness=completeness,
                v_measure=v_measure,
                mloc=mloc,
                labels=labels,
                preds=preds,
                accuracy=accuracy,
                step=i,
                )
            rd = {k:np.array(v) for k,v in rd.items()}
            return rd


        if lne is not None or i == maxIt - 1:
            if (lne == mean_error) or (i == maxIt - 1):
                return prep_return()
        lne = mean_error
```