Evaluation –
System Accuracy

Recommender Systems

Dimitris Sacharidis

flavors of system accuracy

- **rating prediction accuracy**
  - how **close** are the predicted ratings to actual ratings?
  - e.g., system predicts that I would rate a movie with 4.3, whereas my actual rating is 4
- **classification accuracy**
  - are the recommendations **relevant** for me?
  - e.g., system gives me 5 recommendations, among which 4 are interesting
- **ranking accuracy**
  - does a ranked list of recommendations **contain relevant** items?
  - e.g., system ranks 5 items, where the top-3 are interesting

rating prediction accuracy

MAE, RMSE

rating prediction accuracy

how accurate are the predicted ratings?

two ingredients:

- a way to quantify **error** between **predicted** and **actual** rating
    - typically absolute error, or squared error
    - but could be relative error
- a way to **aggregate** individual errors
    - typically the mean

- two examples:
    - (root of) mean of squared errors (R)MSE
    - mean of absolute errors MAE

MAE and RMSE

- the error on the rating of an item by a user is

$$e_{ui} = r_{ui} - \hat{r}_{ui}$$

- where r_{ui} is the actual rating and $\hat{r}_{ui}$ is the predicted rating

- **mean absolute error** (MAE) over a set of ratings R is

$$\frac{1}{|R|} \sum_{r_{ui} \in R} |e_{ui}|$$

- **root mean squared error** (RMSE) over a set of ratings R is

$$\sqrt{\frac{1}{|R|} \sum_{r_{ui} \in R} e_{ui}^2}$$

MAE and RMSE

	user	item	rating	prediction	abs. error	squared error
1	1	1	5	3.5	1.5	2.25
2	1	2	4	5	1	1
3	1	3	5	5	0	0
4	2	1	3	5	2	4
5	2	4	5	4.5	0.5	0.25
6	3	5	4	4.1	0.1	0.01
7	3	4	4	3.9	0.1	0.01
8	3	2	3	3	0	0
9	4	5	4	4.2	0.2	0.04
10	4	6	5	4.8	0.2	0.04
				sums	**5.6**	**7.6**
					MAE	**RMSE**
					0.56	**0.872**

- squared error **penalizes big errors**, downplays small errors
- typically RMSE is used
  - e.g., in Netflix prize

RMSE and matrix factorization

- matrix factorization optimizes for RMSE
- recall that **mean squared error** was the main **optimization goal**

$$J = \sum_{r_{ui} \in R} (r_{ui} - \hat{r}_{ui})^2 + \lambda(\|P\|^2 + \|Q\|^2 + \cdots)$$

classification accuracy

precision, recall

binary classification task

- classify items into two classes: **relevant (R)** (of interest to the target user) and **non-relevant (NR)**
- for implicit feedback, relevant means clicked/purchased
- for explicit feedback (ratings), relevant means a high rating
  - e.g., at least 4 out of 5 stars

item	rating	class
1	5	**R**
2	4	**R**
3	5	**R**
4	3	NR
5	5	**R**
6	4	**R**
7	4	**R**
8	3	NR
9	4	**R**
10	5	**R**

binary classification task

- the recommender predicts the class (R or NR) of items for the target user and **recommends items that are predicted to be relevant**
- (e.g., use any CF technique to predict ratings, and then return only items with predicted rating above 4)

item	rating	class	prediction
1	5	**R**	R
2	4	**R**	R
3	5	**R**	NR
4	3	NR	R
5	5	**R**	NR
6	4	**R**	R
7	4	**R**	R
8	3	NR	NR
9	4	**R**	R
10	5	**R**	NR

true/false positives/negatives

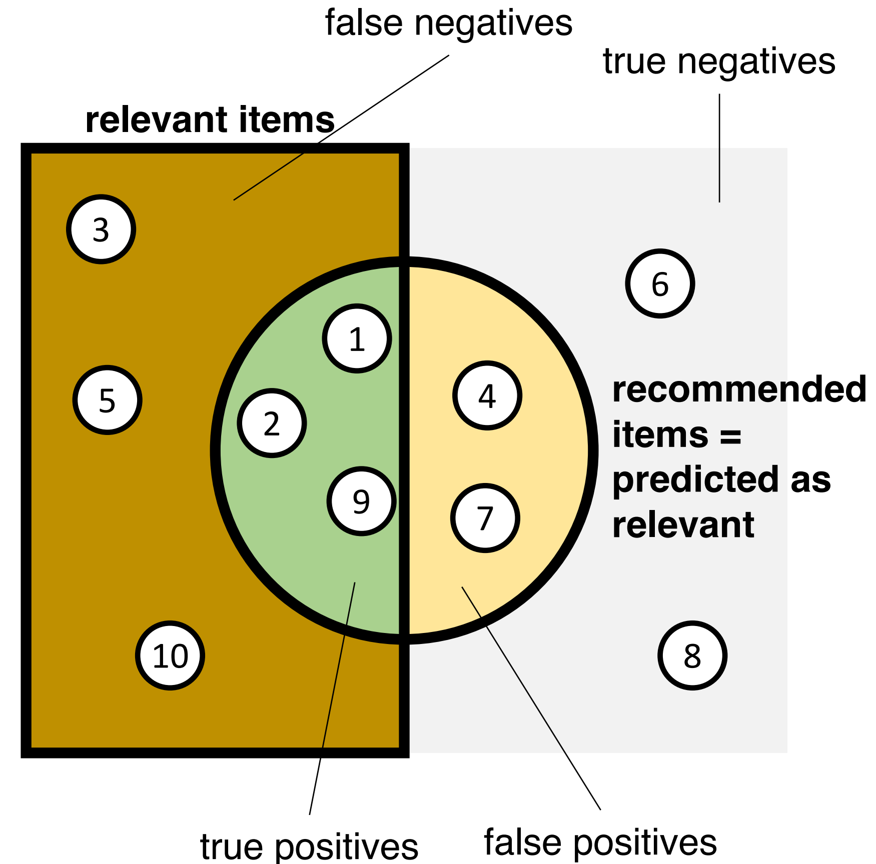
- an item can be:
  - **actually relevant** or **not,** based on the ground truth data
  - **predicted relevant (positive)** or **not (negative),** based on the classifier's decision

- thus 4 cases for any item:
  - **true positive**: correctly predicted relevant
  - **true negative**: correctly predicted as nonrelevant
  - **false positive**: incorrectly predicted relevant
  - **false negative**: incorrectly predicted nonrelevant

		actually	
		R	NR
predicted	R	true positive (TP)	false positive (FP)
	NR	false negative (FN)	true negative (TN)

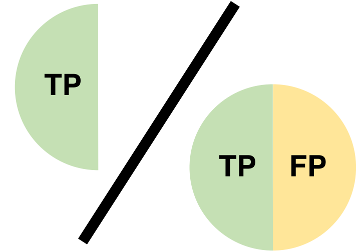
true/false positives/negatives

item	rating	class	prediction	type
1	5	**R**	R	TP
2	4	**R**	R	TP
3	5	**R**	NR	FN
4	3	NR	R	FP
5	5	**R**	NR	FN
6	4	NR	NR	TN
7	4	NR	R	FP
8	3	NR	NR	TN
9	4	**R**	R	TP
10	5	**R**	NR	FN

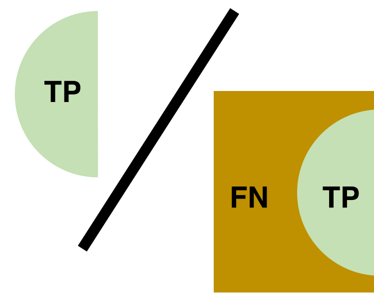


precision and recall

- **precision (P)**: proportion of actually relevant items among recommended
  P = TP/(TP+FP)
  - ideally, the system *only recommends relevant items*



- **recall (R)**: proportion of actually relevant items among all relevant
  R = TP/(TP+FN)
  - ideally, the system *recommends all relevant items*



precision and recall

- what is the easiest way to guarantee 100% recall?
  - what would precision be then?

- can you guarantee 100% precision?

- precision and recall are usually reported together
  - difficult to interpret one without the other
  - this makes comparing methods a bit harder

- F_1 is a single measure combining recall and precision
  - (it's their harmonic mean)

$$F_1 = 2 \cdot \frac{P \cdot R}{P + R} \quad = \frac{2}{\frac{1}{P} + \frac{1}{R}}$$

ranking accuracy

P@k, R@k, AP, NDCG@k

ranks are important

- the output of a recommender is a **ranked list** of items
- a system is effective when **good recommendations appear at the top ranks**
  - why? users only pay attention to the first few recommendations

- how can we capture the effectiveness of the system in terms of the ranked lists it generates?

- measuring prediction accuracy (RMSE) is one way; why?
  - RMSE is an aggregate covering all item errors, whereas we should care more about the errors for the top ranking items

measuring ranking accuracy

what information do we need to measure ranking accuracy?

1) the **ranked list** of recommended items
   - notice, we don't care about the predicted ratings

2) the set of **relevant items (ground truth)**
   - when appropriate, we also consider **relevance scores** for items (e.g., their actual ratings)

precision and recall for ranked lists

- precision/recall of the **entire ranked list** does not take ranks into account
  - treats the ranked list as a set of items; i.e., **rank is irrelevant**

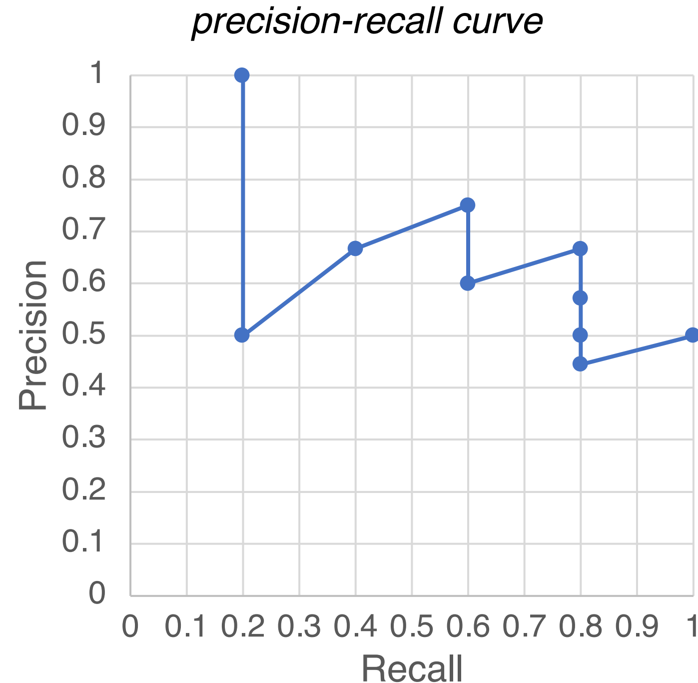
- fix: **cut off** the ranked list at a particular rank, i.e., consider only the **top-k items**
  - what should k be? application/domain specific

- **precision at rank k** (P@k): precision of the top-k recommendations
- **recall at rank k** (R@k): recall of the top-k recommendations

precision-recall curve

- we can compute precision/recall **at all ranks** and draw the **precision-recall curve**

rank	class	P@k	R@k
1	R	1	0.2
2	NR	0.5	0.2
3	R	0.667	0.4
4	R	0.75	0.6
5	NR	0.6	0.6
6	R	0.667	0.8
7	NR	0.571	0.8
8	NR	0.5	0.8
9	NR	0.444	0.8
10	R	0.5	1



average precision (AP)

- AP is a single metric that summarizes precision and recall scores
- AP is the **average of the precision scores at the ranks when a new relevant item is recalled** (i.e., when recall increases)
  - note: AP is **not** the average of all precision scores

rank	class	P@k	R@k
1	**R**	**1**	**0.2**
2	NR	0.5	0.2
3	**R**	**0.667**	**0.4**
4	**R**	**0.75**	**0.6**
5	NR	0.6	0.6
6	**R**	**0.667**	**0.8**
7	NR	0.571	0.8
8	NR	0.5	0.8
9	NR	0.444	0.8
10	**R**	**0.5**	**1**

- bold indicates the ranks when recall increases

AP = (1+0.667+0.75+0.667+0.5)/5 = 0.717

ranks and relevance scores

- ranks are important, but
- **relevance scores**, when available, are also important
  - how relevant are the items at the top ranks
- precision/recall does not care about relevance scores
  - it only cares about **how many** relevant items are at rank k, and not **how much** relevant these items are
- e.g., P@5=0.4, whenever two relevant items appear in the top-5, that is even if
  - both items are highly interesting (have a rating of 5), or
  - both items are slightly interesting (have a rating of 3)

normalized discounted cumulative gain (NDCG)

- NDCG is a metric that cares about **ranks** and **relevance scores**

- a thought experiment to understand the intuition behind NDCG
- consider two ranked lists R1 and R2

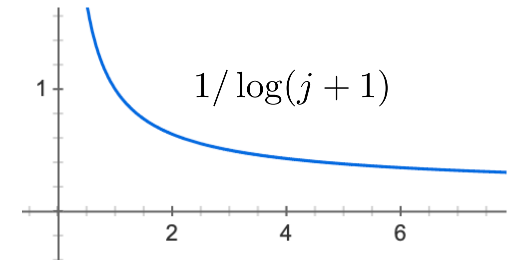
- assume the **same rank** is occupied by **different items**
  - e.g., at rank 1: R1 places an item with rating 5, while R2 an item with rating 4
- clearly R1 utilizes the rank better, but **how much better**?
  - by the ratio of the ratings: 5/4 = 1.25

- assume the **same item** appears at **different ranks**
  - e.g., R1 returns it at rank 1, while R2 at rank 2
- the contribution of the item is higher in R1, but **how much higher**?
  - by the ratio of the log of their ranks: $\log(2+1)/\log(1+1) \approx 1.59$
    - (plus one is to avoid log of 1)

normalized discounted cumulative gain (NDCG)

- consider a ranked list where i_j is the item at rank j
- let r_{u,i_j} denote the relevance of the item to the target user
- the **discounted relevance** of the item due to its rank in the list is

$$\frac{r_{u,i_j}}{\log(j+1)}$$



- the **discounted cumulative gain** (DCG) sums up the discounted relevances up to some rank k

$$\text{DCG@}k = \sum_{j=1}^{k} \frac{r_{u,i_j}}{\log(j+1)}$$

normalized discounted cumulative gain (NDCG)

- the DCG is typically normalized by the ideal DCG, called IDCG

- achieved by the **ideal ranking** that contains all relevant items sorted decreasingly by relevance

$$\mathrm{NDCG@}k = \frac{\mathrm{DCG@}k}{\mathrm{IDCG@}k}$$

normalized discounted cumulative gain (NDCG)

	given ranking			ideal ranking			
rank	**relevance**	**disc. rel.**	**DCG@k**	**relevance**	**disc. rel.**	**IDCG@k**	**NDCG@k**
1	5	5.00	5	5	5.00	5	1
2		0.00	5	5	3.15	8.15	0.613
3	4	2.00	7	4	2.00	10.15	0.689
4	5	2.15	9.15	4	1.72	11.88	0.771
5		0.00	9.15	3	1.16	13.04	0.702
6	4	1.42	10.58		0.00	13.04	0.811
7		0.00	10.58		0.00	13.04	0.811
8		0.00	10.58		0.00	13.04	0.811
9		0.00	10.58		0.00	13.04	0.811
10	3	0.87	11.45		0.00	13.04	0.878

how to evaluate

train and test data

- recommenders require a set of historical feedback data in order to make recommendations
- the data used in this process is called the **train dataset**

- the goal of the accuracy evaluation is to see how well the recommender performs for **unseen** ratings
  - how well the recommender can **generalize** what it has seen in the train dataset

- evaluation should be executed on a distinct **test dataset**

- train and test dataset should not over overlap, but have similar distribution

how to split into train and test data

- typically one has access to a big historical feedback dataset
  - e.g., a set of user-item ratings (MovieLens 1M), a set of ranked lists for users
- this should be split into a train and test dataset
- how to split a dataset into training and test subsets?
- in some cases, the split is already performed
  - e.g., test dataset contains some hard cases that we care for
- in most cases, you perform the split
  - decide on a split ratio (e.g., 80:20 or 90:10) and randomly partition the dataset

train	test

how to split into train and test data

- a recommender is **performing well** when it makes good recommendations for **all users**
- so split should be ***user-aware***
  - and not just a random partition of known ratings

- assume a target split ratio 80:20
- then, for each user:
  - randomly select 80% of historical feedback and insert into train

how to report evaluation metrics

- for rating prediction accuracy:
  - report the metric (MAE or RMSE) over **all ratings** in the test set

- for classification and ranking accuracy:
  - compute the metric (P, R, NDCG@*k*, etc.) individually **for each user** on her/his relevant items in the test set
  - compute the **mean** of the metric **across all users**
    - sidenote: when the metric is average precision (AP), the average across users is called *mean average precision* (MAP)

train, dev, and test data

- model-based recommenders also have hyperparameters (variables that are not learned during training)
  - e.g., number of features *k* in matrix factorization, regularization strength, learning rate

- how should we set hyperparameter values?
  - try multiple values (e.g., by grid search, random search)
  - and select those that works best on **dev data** (development or validation data)
- train, dev, and test are disjoint
  - typical ratio is 80:10:10

train dev test

how to produce dev data

- standard approach is to **split the non-test data** into train and dev subsets
- simplest way is **holdout**: randomly split (but user-aware) trying to achieve desired ratio
- typically used however is ***k*-fold cross validation**

k-fold cross validation

- it considers *k* partitions (called folds) of train-dev data
  - evaluation metrics are reported as average across these *k* folds

- first, randomly divide non-test data into *k* roughly equal parts

- for each fold, choose one part as the dev data, and the other *k*-1 parts as train

	train			dev
fold 1	2	3	4	1
fold 2	1	3	4	2
fold 3	1	2	4	3
fold 4	1	2	3	4