Introduction to Recommender System

Recommender systems apply data analysis & ML techniques to help users decide the items they would like to purchase at E-Commerce sites/Books by p**roducing a predicted likeliness score** or a list of top N recommended items for a given user.
Recommender systems typically produce a list of recommendations in one of two ways – through collaborative and content-based filtering.
Collaborative filtering approaches build a model from a user's past behaviour.
Content-based filtering approaches utilize a series of discrete characteristics of an item in order to recommend additional items with similar properties.

Introduction to GraphLab(here)

GraphLab is a parallel framework for machine learning. Data structure of GraphLab can handle large data sets which result into scalable machine learning.

Data structure of GraphLab:

- SFrame: It is an efficient disk-based tabular data structure not limited by RAM. It helps to scale analysis and data processing to handle large data set (Tera byte). It has similar syntax like pandas or R data frames. Each column is an SArray, which is a series of elements stored on disk. This makes SFrames disk based.
- SGraph: Graph helps us to understand networks by analyzing relationships between pair of items. Each item is represented by a vertex in the graph. Relationship between items is represented by edges. In GraphLab, to perform a graph-oriented data analysis, it uses SGraph object.

GraphLab Canvas

Data exploration and visualization with GraphLab Canvas. GraphLab Canvas is a browser-based interactive GUI which allows you to explore tabular data, summary statistics and bi-variate plots. Using this feature, you spend less time coding for data exploration.

Modeling

GraphLab has various toolkits to deliver easy and fast solution for ML problems. It allows you to perform various modeling exercise (regression, classification, clustering) in fewer lines of code. You can work on problems like recommendation system, churn prediction, sentiment analysis, image analysis.

Getting Started

Importing libraries

After installing GraphLab from here we can start by importing the libraries.

```
import graphlab
```

Run graphlab.get_dependencies() to download and install the required dependencies. This needs to be done only the first time, after that, the next time we try to run a code, the dependencies will already be present, hence no need to run graphlan.get_dependencies() again.

Loading the data set

We can load the data set from external source and convert it to SFrame so that GraphLab can work with it. SFrame is the tabular data structure available in GraphLab. SFrames can import data in various formats (CSV, JSON, Apache Avro, ODBC ..). A common data format is comma separated value (csv) file.

```
train_csv_data= pd.read_csv('path/training_data.csv', sep=',')
test_csv_data=pd.read_csv('path/test_data.csv', sep=',')
Bookdata_csv = pd.read_csv('path/mergefullnewdata.csv', sep=',')
train_data = graphlab.SFrame(train_csv_data)
test_data= graphlab.SFrame(test_csv_data)
book_data= graphlab.SFrame(Bookdata_csv)
```

Here, we are first reading the csv using Pandas, and then converting it to an SFrame data structure.

Visualizing the data set

Data in an SFrame can be visualized with SFrame.show(). When we run command SFrame.show(), it returns a URL which redirects to GraphLab Canvas.

Recommender Models in GraphLab(here)

A recommender system allows you to provide personalized recommendations to users. With this toolkit, you can train a model based on past interaction data and use that model to make recommendations.

Building a model(here)

GraphLab Create provides a method graphlab.recommender.create() that will automatically choose an appropriate model for your data set or loaded from a previously saved model using graphlab.load_model().
The input data must be an SFrame with a column containing user ids, a column containing item ids, and optionally a column containing target values such as movie ratings, etc. When a target is not provided (as is the case in implicit feedback settings), then a collaborative filtering model based on item-item similarity is returned.
First we create a random split of the data to produce a validation set that can be used to evaluate the model.Data is split into 70% training and 30% testing.

```
train_data, test_data = cv.train_test_split(rating, test_size=0.3)
```

The easiest way to choose a model is to let GraphLab Create choose your model for you. This is done by simply using the default recommender.create function, which chooses the model based on the data provided to it. As an example, the following code creates a basic item similarity model and then generates recommendations for each user in the dataset:

```
model = gl.recommender.create(training_data, 'userId', 'bookId')
```

Using the default create method provides an excellent way to quickly get a recommender model up and running, but in many cases it's desirable to have more control over the process.

Effectively choosing and tuning a recommender model is best done in two stages. The first stage is to match the type of data up with the correct model or models, and the second stage is to correctly evaluate and tune the model(s) and assess their accuracy. Sometimes one model works better and sometimes another, depending on the data set.

Now that you have a model, you can make recommendations:

```
results = model.recommend()
```

Evaluating model performance

With recommender systems, we can evaluate models using two different metrics, RMSE and precision-recall. RMSE measures how well the model predicts the score of the user, while precision-recall measures how well the recommend() function recommends items that the user also chooses. For example, the best RMSE value is when the model exactly predicts the value of all the ratings in the test set. Similarly, the best precision-recall value is when the user has 5 items in the test set and

recommend() recommends exactly those 5 items. While both can be important depending on the type of data and desired task, precision-recall is often more useful in evaluating how well a recommender system will perform in practice.
If the model is trained to predict a particular target, the default metric used for model comparison is root-mean-squared error (RMSE). Suppose y and $\widehat{y}$ are vectors of length N, where y contains the actual ratings and $\widehat{y}$ the predicted ratings. Then the RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}.$$

If the model was not trained on a target column, the default metrics for model comparison are precision and recall. Let p_k be a vector of the k highest ranked recommendations for a particular user, and let a be the set of items for that user in the groundtruth dataset. The “precision at cutoff k” is defined as

$$P(k) = \frac{|a \cap p_k|}{k}$$

while “recall at cutoff k” is defined as

$$R(k) = \frac{|a \cap p_k|}{|a|}$$

Comparing the models(here)

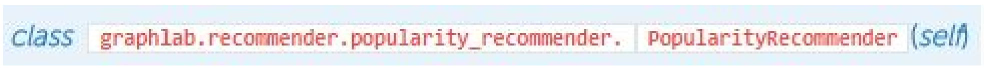
`graphlab. show_comparison (model_comp, models)`

Visualizes the model comparison result (compare) in Canvas.
Parameters:
model_comp : SFrame
    The dataset to use for model evaluation.

models : list[models]
    List of trained models of single type. Currently, only Recommender is supported.

Popularity based recommender model(here)

Popularity based model, i.e. the one where all the users have same recommendation based on the most popular choices. We'll use the graphlab recommender functions popularity_recommender for this.



```
class graphlab.recommender.popularity_recommender.PopularityRecommender(self)
```

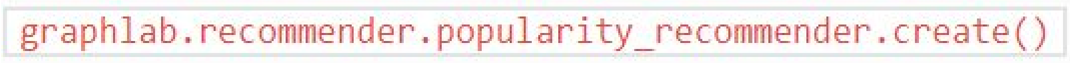
The Popularity Model ranks an item according to its overall popularity.

When making recommendations, the items are scored by the number of times it is seen in the training set. The item scores are the same for all users. Hence the recommendations are not tailored for individuals.

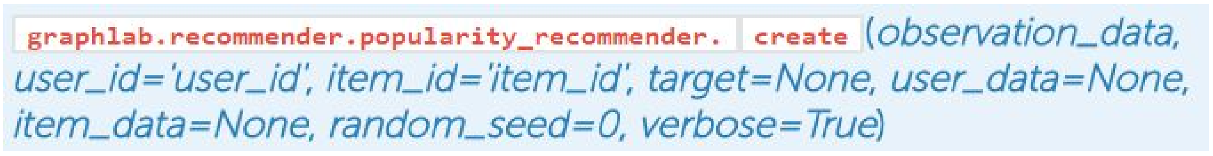
The Popularity Recommender is simple and fast and provides a reasonable baseline. It can work well when observation data is sparse. It can be used as a "background" model for new users.

Creating a Popularity Recommender

This model is constructed using the method

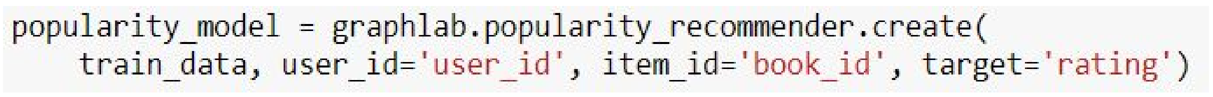


```
graphlab.recommender.popularity_recommender.create()
```



```
graphlab.recommender.popularity_recommender.create(observation_data, user_id='user_id', item_id='item_id', target=None, user_data=None, item_data=None, random_seed=0, verbose=True)
```

As in our book recommendation as:



```
popularity_model = graphlab.popularity_recommender.create(
    train_data, user_id='user_id', item_id='book_id', target='rating')
```

Arguments:

- train_data: the SFrame which contains the required data
- user_id: the column name which represents each user ID
- item_id: the column name which represents each item to be recommended
- target: the column name representing scores/ratings given by the user

Recommendations based on the Popularity model

After creating the Popularity Model, we can do the recommendations as:

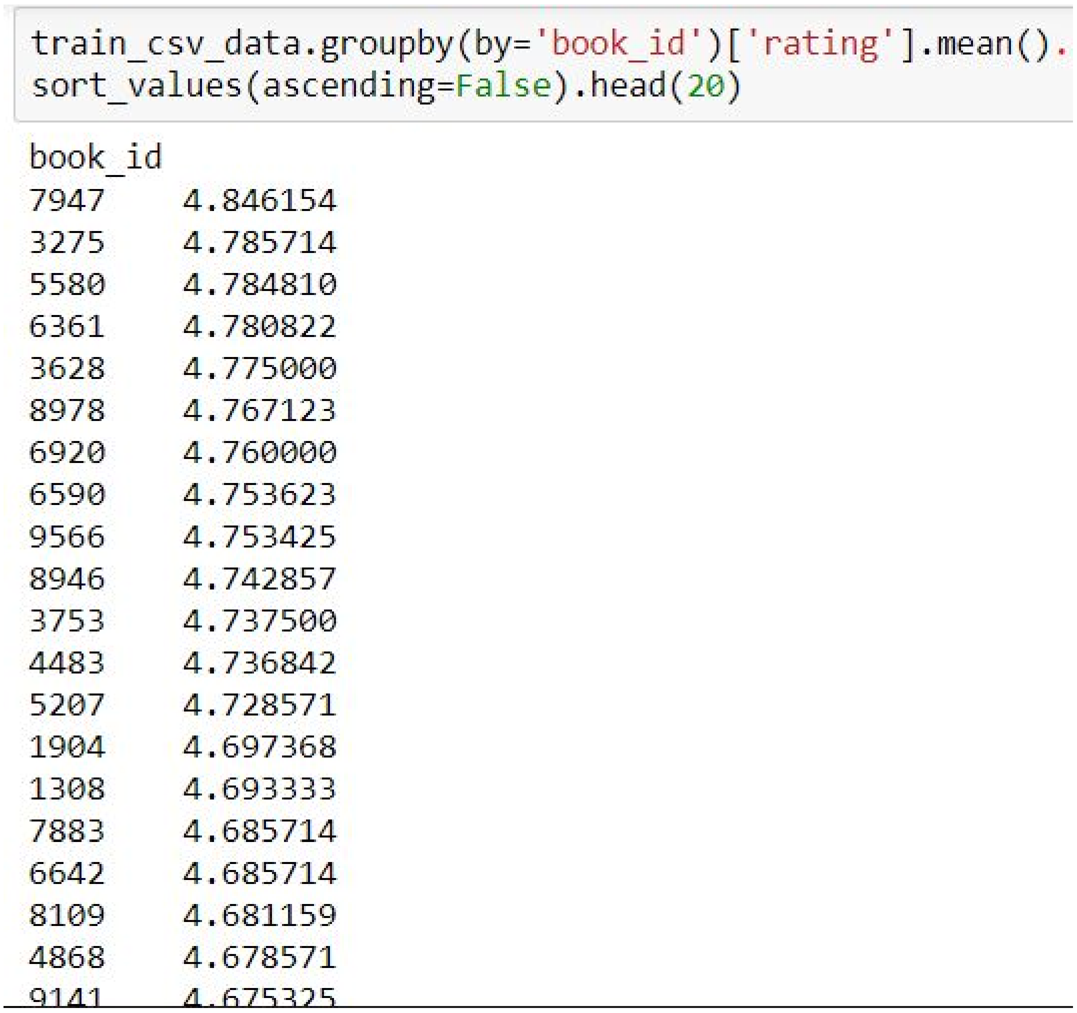
```
popularity_recomm = popularity_model.recommend(users=range(1,6),k=5)
```

```
popularity_recomm.print_rows(num_rows=25)
```

The output we get for the recommendations are:

```
+---------+---------+---------------+------+
| user_id | book_id |     score     | rank |
+---------+---------+---------------+------+
|    1    |   7947  | 4.84615384615 |  1   |
|    1    |   3275  | 4.78571428571 |  2   |
|    1    |   5580  | 4.78481012658 |  3   |
|    1    |   6361  | 4.78082191781 |  4   |
|    1    |   3628  |     4.775     |  5   |
|    2    |   7947  | 4.84615384615 |  1   |
|    2    |   3275  | 4.78571428571 |  2   |
|    2    |   5580  | 4.78481012658 |  3   |
|    2    |   6361  | 4.78082191781 |  4   |
|    2    |   3628  |     4.775     |  5   |
|    3    |   7947  | 4.84615384615 |  1   |
|    3    |   3275  | 4.78571428571 |  2   |
|    3    |   5580  | 4.78481012658 |  3   |
|    3    |   6361  | 4.78082191781 |  4   |
|    3    |   3628  |     4.775     |  5   |
|    4    |   7947  | 4.84615384615 |  1   |
|    4    |   3275  | 4.78571428571 |  2   |
|    4    |   5580  | 4.78481012658 |  3   |
|    4    |   6361  | 4.78082191781 |  4   |
|    4    |   3628  |     4.775     |  5   |
|    5    |   7947  | 4.84615384615 |  1   |
|    5    |   3275  | 4.78571428571 |  2   |
|    5    |   5580  | 4.78481012658 |  3   |
|    5    |   6361  | 4.78082191781 |  4   |
|    5    |   3628  |     4.775     |  5   |
+---------+---------+---------------+------+
```

The recommendations for all users are same. This can be verified by checking the movies with highest mean recommendations in our train_csv_data data set.

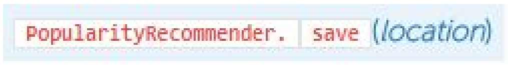


```
train_csv_data.groupby(by='book_id')['rating'].mean().
sort_values(ascending=False).head(20)
```

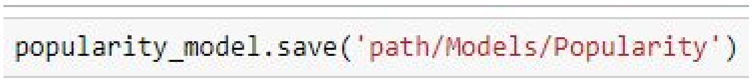
```
book_id
7947    4.846154
3275    4.785714
5580    4.784810
6361    4.780822
3628    4.775000
8978    4.767123
6920    4.760000
6590    4.753623
9566    4.753425
8946    4.742857
3753    4.737500
4483    4.736842
5207    4.728571
1904    4.697368
1308    4.693333
7883    4.685714
6642    4.685714
8109    4.681159
4868    4.678571
9141    4.675325
```

This confirms that all the recommended books have an average rating of 5, i.e. all the users who read the book gave a top rating. Thus we can see that our popularity system works as expected.

Saving the Popularity Model



```
PopularityRecommender. save (location)
```



```
popularity_model.save('path/Models/Popularity')
```

Parameter:
Location: string type:-Target destination for the model. Can be a local path or remote URL.

The saved model can be later loaded back using the load_model() method.

Collaborative Filtering recommender model(here)

Works in 2 steps:

1. Find similar items by using a similarity metric
2. For a user, recommend the items most similar to the items (s)he already likes

This is done by making an item-item matrix in which we keep a record of the pair of items which were rated together.
In this case, an item are books. Once we have the matrix, we use it to determine the best recommendations for a user based on the books he has already rated.
This model first computes the similarity between items using the observations of users who have interacted with both items. Given a similarity between item i and j, S(i,j), it scores an item j for user u using a weighted average of the user's previous observations Iu.
There are three choices of similarity metrics to use: 'jaccard', 'cosine' and 'pearson'.
Jaccard similarity is used to measure the similarity between two set of elements. In the context of recommendation, the Jaccard similarity between two items is computed as

$$\mathrm{JS}(i,j) = \frac{|U_i \cap U_j|}{|U_i \cup U_j|}$$

where Ui is the set of users who rated item i. Jaccard is a good choice when one only has implicit feedbacks of items (e.g., people rated them or not), or when one does not care about how many stars items received.
If one needs to compare the ratings of items, Cosine and Pearson similarity are recommended.
The Cosine similarity between two items is computed as

$$\mathrm{CS}(i,j) = \frac{\sum_{u \in U_{ij}} r_{ui} r_{uj}}{\sqrt{\sum_{u \in U_i} r_{ui}^2} \sqrt{\sum_{u \in U_j} r_{uj}^2}}$$

where Ui is the set of users who rated item i, and Uij is the set of users who rated both items i and j. A problem with Cosine similarity is that it does not consider the differences in the mean and variance of the ratings made to items i and j.
Another popular measure that compares ratings where the effects of means and variance have been removed is Pearson Correlation similarity:

$$\mathrm{PS}(i,j) = \frac{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)(r_{uj} - \bar{r}_j)}{\sqrt{\sum_{u \in U_{ij}} (r_{ui} - \bar{r}_i)^2} \sqrt{\sum_{u \in U_{ij}} (r_{uj} - \bar{r}_j)^2}}$$

The predictions of items depend on whether target is specified. When the target is absent, a prediction for item j is made via

$$y_{uj} = \frac{\sum_{i \in I_u} \mathrm{SIM}(i,j)}{|I_u|}$$

Otherwise, predictions for jaccard and cosine similarities are made via

$$y_{uj} = \frac{\sum_{i \in I_u} \mathrm{SIM}(i,j) r_{ui}}{\sum_{i \in I_u} \mathrm{SIM}(i,j)}$$

Predictions for pearson similarity are made via

$$y_{uj} = \bar{r}_j + \frac{\sum_{i \in I_u} \mathrm{SIM}(i,j)(r_{ui} - \bar{r}_i)}{\sum_{i \in I_u} \mathrm{SIM}(i,j)}$$

Creating a Collaborative Filtering model

This model is constructed using the method

```
graphlab.recommender.item_similarity_recommender. create (
```

`graphlab.recommender.item_similarity_recommender. create` *(observation_data, user_id='user_id', item_id='item_id', target=None, user_data=None, item_data=None, nearest_items=None, similarity_type='jaccard', threshold=0.001, only_top_k=64, verbose=True, target_memory_usage=8589934592, **kwargs)*

Parameters:
observation_data : test_data
user_id : user_id
item_id : book_id
target : rating
similarity_type: pearson
only_top_k=100
training_method={'auto', 'dense', 'sparse', 'nn', 'nn:dense', 'nn:sparse'}

It is applied to our book recommendation system for training_method="auto" as:

```
#to create the model training_method=auto
item_sim_model_auto = graphlab.item_similarity_recommender.
create(test_data, user_id='user_id', item_id='book_id',
target='rating', similarity_type='pearson',only_top_k=100,training_method='auto')
```

observation_data : SFrame, test_data
The dataset to use for training the model. It must contain a column of user ids and a column of item ids. Each row represents an observed interaction between the user and the item. The (user, item) pairs are stored with the model so that they can later be excluded from recommendations if desired. It can optionally contain a target ratings column. All other columns are interpreted by the underlying model as side features for the observations.
The user id and item id columns must be of type 'int' or 'str'. The target column must be of type 'int' or 'float'.

user_id : string, optional
The name of the column in observation_data that corresponds to the user id.

item_id : string, optional
The name of the column in observation_data that corresponds to the item id.

target : string, optional
The observation_data can optionally contain a column of scores representing ratings given by the users. If present, the name of this column may be specified variables target.

similarity_type : {‘jaccard’, ‘cosine’, ‘pearson’}, optional
Similarity metric to use. Default: ‘jaccard’.

only_top_k : int, optional
Number of similar items to store for each item. Default value is 64. Decreasing this decreases the amount of memory required for the model, but may also decrease the accuracy.

training_method : optional.
The internal processing is done with a combination of nearest neighbor searching, dense tables for tracking item-item similarities, and sparse item-item tables. If ‘auto’ is chosen (default), then the estimated computation time is estimated for each, and the computation balanced between the methods in order to minimize training time given the target memory usage. This allows the user to force the use of one of these methods. All should give equivalent results; the only difference would be training time. Possible values are {‘auto’, ‘dense’, ‘sparse’, ‘nn’, ‘nn:dense’, ‘nn:sparse’}. ‘dense’ uses a dense matrix to store item-item interactions as a lookup, and may do multiple passes to control memory requirements. ‘sparse’ does the same but with a sparse lookup table; this is better if the data has many infrequent items. “nn” uses a brute-force nearest neighbors search. “nn:dense” and “nn:sparse” use nearest neighbors for the most frequent items, and either sparse or dense matrices for the remainder. “auto” chooses the method predicted to be the fastest based on the properties of the data.

Similarly, for other training_method it is applied as:

```
#to create the model training_method=dense
item_sim_model_dense = graphlab.item_similarity_recommender.
create(train_data, user_id='user_id', item_id='book_id',
target='rating', similarity_type='pearson',only_top_k=100,training_method='dense')
```

```
#to create the model training_method=sparse
item_sim_model_sparse = graphlab.item_similarity_recommender.
create(train_data, user_id='user_id', item_id='book_id',
target='rating', similarity_type='pearson',only_top_k=100,training_method='sparse')
```

```
#to create the model training_method=nnsparse
item_sim_model_nnsparse = graphlab.item_similarity_recommender.
create(train_data, user_id='user_id', item_id='book_id',
target='rating', similarity_type='pearson',only_top_k=100,training_method='nn:sparse')
```

```
#to create the model training_method=nndense
item_sim_model_nndense = graphlab.item_similarity_recommender.
create(train_data, user_id='user_id', item_id='book_id',
target='rating', similarity_type='pearson',only_top_k=100,training_method='nn:dense')
```

Comparing the Collaborative Filtering model(here)

Compare the prediction or recommendation performance of recommender models on a common test dataset.

Models that are trained to predict ratings are compared separately from models that are trained without target ratings. The ratings prediction models are compared on root-mean-squared error, and the rest are compared on precision-recall.

`graphlab.recommender.util.` `compare_models` *(dataset, models, model_names=None, user_sample=1.0, metric='auto', target=None, exclude_known_for_precision_recall=True, make_plot=False, verbose=True, **kwargs)*

Here, we will compare the different collaborative models we created earlier:

```
graphlab.recommender.util.compare_models(
    test_data,[item_sim_model_auto,item_sim_model_dense,item_sim_model_sparse,
    item_sim_model_nnsparse,item_sim_model_nndense],model_names=
    ["item_sim_model_auto","item_sim_model_dense","item_sim_model_sparse",
     "item_sim_model_nnsparse","item_sim_model_nndense"])
```

Parameters:
dataset: test_data
models : [item_sim_model_auto,item_sim_model_dense,item_sim_model_sparse,
  item_sim_model_nnsparse,item_sim_model_nndense]
model_names :
["item_sim_model_auto","item_sim_model_dense","item_sim_model_sparse",
  "Item_sim_model_nnsparse","item_sim_model_nndense"]

Comparision on GraphLab Canvas

```
graphlab.show_comparison(model_performance,
[item_sim_model_auto,item_sim_model_dense,item_sim_model_sparse,
 item_sim_model_nndense,item_sim_model_nnsparse])
```

Recommendations based on the Collaborative model(here)

Now, we try to make recommendations for this model.

`ItemSimilarityRecommender.` `recommend` *(users=None, k=10, exclude=None, items=None, new_observation_data=None, new_user_data=None, new_item_data=None, exclude_known=True, diversity=0, random_seed=None, verbose=True)*

Recommend the `K` highest scored items for each user. The number of recommendations to generate for each user.

Parameters:
users : SArray, SFrame, or list, optional
Users or observation queries for which to make recommendations. By default, recommendations are returned for all users present when the model was trained.

k : int, optional
The number of recommendations to generate for each user.

items : SArray, SFrame, or list, optional
Restricts the items from which recommendations can be made. By default, recommendations are made from all items present when the model was trained.

new_observation_data : SFrame, optional
new_observation_data gives additional observation data to the model, which may be used by the models to improve score and recommendation accuracy. Must be in the same format as the observation data passed to create. How this data is used varies by model.
new_user_data : SFrame, optional
new_user_data may give additional user data to the model. If present, scoring is done with reference to this new information. If there is any overlap with the side information present at training time, then this new side data is preferred. Must be in the same format as the user data passed to create.

new_item_data : SFrame, optional
new_item_data may give additional item data to the model. If present, scoring is done with reference to this new information. If there is any overlap with the side information present at training time, then this new side data is preferred. Must be in the same format as the item data passed to create.

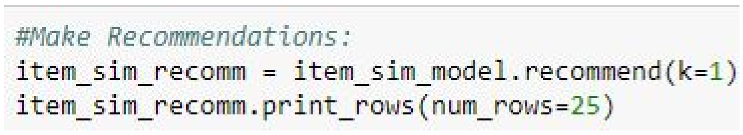
exclude : SFrame, optional
An SFrame of user / item pairs. The column names must be equal to the user and item columns of the main data, and it provides the model with user/item pairs to exclude from the recommendations. These user-item-pairs are always excluded from the predictions, even if exclude_known is False.

exclude_known : bool, optional
By default, all user-item interactions previously seen in the training data, or in any new data provided using new_observation_data.., are excluded from the recommendations. Passing in exclude_known = False overrides this behavior.

diversity : non-negative float, optional
If given, then the recommend function attempts chooses a set of k items that are both highly scored and different from other items in that set. It does this by first retrieving k*(1+diversity) recommended items, then randomly choosing a diverse set from these items. Suggested values for diversity are between 1 and 3.

random_seed : int, optional
If diversity is larger than 0, then some randomness is used; this controls the random seed to use for randomization. If None, will be different each time.

Applied to our model, we have:



```
#Make Recommendations:
item_sim_recomm = item_sim_model.recommend(k=1)
item_sim_recomm.print_rows(num_rows=25)
```

We have, k=1, hence, we will get only 1 recommendation per user.

For training_method=”auto” we have the recommendation code as:

```
r = item_sim_model_auto.recommend(new_user_data=test_data)
r.print_rows(num_rows=50)
```

For training_method="dense" we have the recommendation code as:

```
recom=item_sim_model_dense.recommend()
recom.print_rows(num_rows=50)
```

If the value for arguments are not explicitly give, we will have default values as specified by the GraphLab library.

Result of ItemSimilarityRecommender.recommend:
out : SFrame
A SFrame with the top ranked items for each user. The columns are: user_id, item_id, score, and rank, where user_id and item_id match the user and item column names specified at training time. The rank column is between 1 and k and gives the relative score of that item. The value of score depends on the method used for recommendations.

Predicting the result of the model**(here)**

Return a score prediction for the user ids and item ids in the provided data set.

```
ItemSimilarityRecommender. predict (dataset, new_observation_data=None, new_user_data=None, new_item_data=None)
```

Applied to our training_method="dense" model we have:

```
item_sim_model_dense.predict(test_data)
```

Parameters:

dataset : SFrame
Dataset in the same form used for training.

new_observation_data : SFrame, optional
new_observation_data gives additional observation data to the model, which may be used by the models to improve score accuracy. Must be in the same format as the observation data passed to create. How this data is used varies by model.

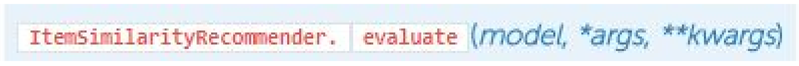
new_user_data : SFrame, optional
new_user_data may give additional user data to the model. If present, scoring is done with reference to this new information. If there is any overlap with the side information present at training time, then this new side data is preferred. Must be in the same format as the user data passed to create.

new_item_data : SFrame, optional
new_item_data may give additional item data to the model. If present, scoring is done with reference to this new information. If there is any overlap with the side information present at

training time, then this new side data is preferred. Must be in the same format as the item data passed to create.

Result of ItemSimilarityRecommender.predict:
out : SArray
An SArray with predicted scores for each given observation predicted by the model.

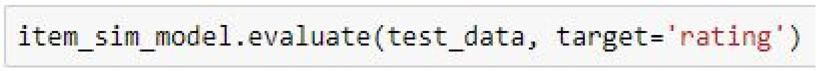
Evaluating the ItemSimilarityModel(here)



```
ItemSimilarityRecommender. evaluate (model, *args, **kwargs)
```

Evaluate the model’s ability to make rating predictions or recommendations.

Applied to our model as:



```
item_sim_model.evaluate(test_data, target='rating')
```

Parameters:

dataset : SFrame
An SFrame that is in the same format as provided for training.

metric : str, {‘auto’, ‘rmse’, ‘precision_recall’}, optional
Metric to use for evaluation. The default automatically chooses ‘rmse’ for models trained with a target, and ‘precision_recall’ otherwise.

exclude_known_for_precision_recall : bool, optional
A useful option for evaluating precision-recall. Recommender models have the option to exclude items seen in the training data from the final recommendation list. Set this option to True when evaluating on test data, and False when evaluating precision-recall on training data.

target : str, optional
The name of the target column for evaluating rmse. If the model is trained with a target column, the default is to using the same column. If the model is trained without a target column and metric is set to ‘rmse’, this option must provided by user.

Result of ItemSimilarityRecommender.evaluate:

out : SFrame or dict
Results from the model evaluation procedure. If the model is trained on a target (i.e. RMSE is the evaluation criterion), a dictionary with three items is returned: items rmse_by_user and

rmse_by_item are SFrames with per-user and per-item RMSE, while rmse_overall is the overall RMSE (a float). If the model is trained without a target (i.e. precision and recall are the evaluation criteria) an SFrame is returned with both of these metrics for each user at several cutoff values.

```
ItemSimilarityRecommender. evaluate_rmse (dataset, target)
```

Evaluate the prediction error for each user-item pair in the given data set.
Applied to our model as:

```
item_sim_model.evaluate_rmse(test_data, target='rating')
```

Parameters:
dataset : SFrame
An SFrame in the same format as the one used during training.

target : str
The name of the target rating column in dataset.

Result of ItemSimilarityRecommender.evaluate_rmse:

out : dict
A dictionary with three items: ‘rmse_by_user’ and ‘rmse_by_item’, which are SFrames containing the average rmse for each user and item, respectively; and ‘rmse_overall’, which is a float.

Content based recommender model(here)

A content based recommender works with data that the user provides, either explicitly (rating) or implicitly (clicking on a link). Based on that data, a user profile is generated, which is then used to make suggestions to the user. As the user provides more inputs or takes actions on the recommendations, the engine becomes more and more accurate.
A recommender based on the similarity between item content rather using user interaction patterns to compute similarity.
This model first computes the similarity between items using the content of each item. The similarity score between two items is calculated by first computing the similarity between the item data for each column, then taking a weighted average of the per-column similarities to get the final similarity. The recommendations are generated according to the average similarity of a candidate item to all the items in a user’s set of rated items.

```
class graphlab.recommender.item_content_recommender. ItemContentRecommender (self)
```

Creating Content based recommender(here)

```
graphlab.recommender.item_content_recommender. create (item_data, item_id, observation_data=None, user_id=None, target=None, weights='auto', similarity_metrics='auto', item_data_transform='auto', max_item_neighborhood_size=64, verbose=True)
```

Create a content-based recommender model in which the similarity between the items recommended is determined by the content of those items rather than learned from user interaction data.
If given, item_data_transform represents a feature_engineering transformer that is applied to the item_data. If item_data_transform is ‘auto’ (default), then an AutoVectorizer transformer is created and applied to item_data. In both cases, the resulting transformer is available in the resulting recommender.
The similarity score between two items is calculated by first computing the similarity between the item data for each column, then taking a weighted average of the per-column similarities to get the final similarity. The recommendations are generated according to the average similarity of a candidate item to all the items in a user’s set of rated items.

Applied to our model as:

```
content_based_model=graphlab.recommender.item_content_recommender.create(
    book_data,item_id='book_id',observation_data=train_data,user_id='user_id',
    target='rating',similarity_metrics='pearson')
```

Parameters:

item_data : SFrame
An SFrame giving the content of the items to use to learn the structure of similar items. The SFrame must have one column that matches the name of the item_id; this gives a unique identifier that can then be used to make recommendations. The rest of the columns are then used in the distance calculations below.

item_id : string
The name of the column in item_data (and observation_data, if given) that represents the item ID.

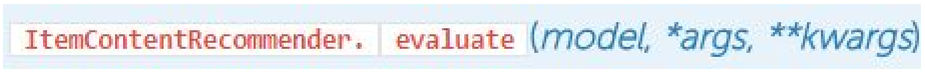
observation_data : None (optional)
An SFrame giving user and item interaction data. This information is stored in the model, and the recommender will recommend the items with the most similar content to the items that were present and/or highly rated for that user.

user_id : None (optional)
If observation_data is given, then this specifies the column name corresponding to the user identifier.

target : str, (optional)
The name of the target column for evaluating rmse. If the model is trained with a target column, the default is to using the same column. If the model is trained without a target column and metric is set to ‘rmse’, this option must provided by user.

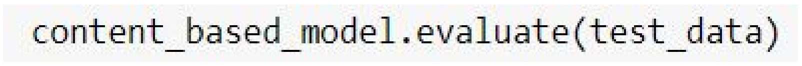
target_id : None (optional)
If observation_data is given, then this specifies the column name corresponding to the target or rating.

Evaluating Content based recommender(here)



```
ItemContentRecommender. evaluate (model, *args, **kwargs)
```

Applied to our code as:



```
content_based_model.evaluate(test_data)
```

Parameters:

dataset : SFrame
An SFrame that is in the same format as provided for training.

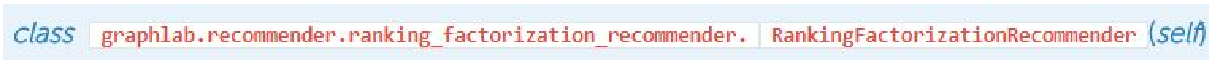
metric : str, {‘auto’, ‘rmse’, ‘precision_recall’}, optional
Metric to use for evaluation. The default automatically chooses ‘rmse’ for models trained with a target, and ‘precision_recall’ otherwise.

**kwargs
When metric is set to ‘precision_recall’, these parameters are passed on to evaluate_precision_recall().

Result of ItemContentRecommender.evaluate
out : SFrame or dict

Results from the model evaluation procedure. If the model is trained on a target (i.e. RMSE is the evaluation criterion), a dictionary with three items is returned: items *rmse_by_user* and *rmse_by_item* are SFrames with per-user and per-item RMSE, while *rmse_overall* is the overall RMSE (a float). If the model is trained without a target (i.e. precision and recall are the evaluation criteria) an `SFrame` is returned with both of these metrics for each user at several cutoff values.

Ranking Factorization Model(here)



```
class graphlab.recommender.ranking_factorization_recommender. RankingFactorizationRecommender (self)
```

A RankingFactorizationRecommender learns latent factors for each user and item and uses them to rank recommended items according to the likelihood of observing those (user, item) pairs. This is commonly desired when performing collaborative filtering for implicit feedback datasets or datasets with explicit ratings for which ranking prediction is desired.

RankingFactorizationRecommender contains a number of options that tailor to a variety of datasets and evaluation metrics, making this one of the most powerful models in the GraphLab Create recommender toolkit.

Model Details

RankingFactorizationRecommender trains a model capable of predicting a score for each possible combination of users and items. The internal coefficients of the model are learned from known scores of users and items. Recommendations are then based on these scores. In the two factorization models, users and items are represented by weights and factors. These model coefficients are learned during training. Roughly speaking, the weights, or bias terms, account for a user or item's bias towards higher or lower ratings. For example, an item that is consistently rated highly would have a higher weight coefficient associated with them. Similarly, an item that consistently receives below average ratings would have a lower weight coefficient to account for this bias.
The factor terms model interactions between users and items. For example, if a user tends to love romance movies and hate action movies, the factor terms attempt to capture that, causing the model to predict lower scores for action movies and higher scores for romance movies. Learning good weights and factors is controlled by several options outlined below. More formally, the predicted score for user i on item j is given by

$$\text{score}(i,j) = \mu + w_i + w_j + \mathbf{a}^T\mathbf{x}_i + \mathbf{b}^T\mathbf{y}_j + \mathbf{u}_i^T\mathbf{v}_j,$$

where μ is a global bias term, wiwi is the weight term for user i, wj is the weight term for item j, xi and yj are respectively the user and item side feature vectors, and a and b are respectively the weight vectors for those side features. The latent factors, which are vectors of length num_factors, are given by ui and vj.

Training the model

The model is trained using Stochastic Gradient Descent [sgd] with additional tricks [Bottou] to improve convergence. The optimization is done in parallel over multiple threads. This procedure is inherently random, so different calls to create() may return slightly different models, even with the same random_seed.
In the explicit rating case, the objective function we are optimizing for is:

$$\min_{\mathbf{w},\mathbf{a},\mathbf{b},\mathbf{V},\mathbf{U}} \frac{1}{|\mathcal{D}|} \sum_{(i,j,r_{ij})\in\mathcal{D}} \mathcal{L}(\text{score}(i,j), r_{ij}) + \lambda_1(\|\mathbf{w}\|_2^2 + ||\mathbf{a}||_2^2 + ||\mathbf{b}||_2^2) + \lambda_2\left(\|\mathbf{U}\|_2^2 + \|\mathbf{V}\|_2^2\right)$$

where D is the observation dataset, rij is the rating that user ii gave to item j, U=(u1,u2,...) denotes the user's latent factors and V=(v1,v2,...)denotes the item latent factors. The loss function L(y^,y) is (y^−y)2 by default. Λ1 denotes the linear_regularization parameter and λ2 the regularization parameter.
When ranking_regularization is nonzero, then the equation above gets an additional term.

$$\min_{\mathbf{w},\mathbf{a},\mathbf{b},\mathbf{V},\mathbf{U}} \frac{1}{|\mathcal{D}|} \sum_{(i,j,r_{ij})\in\mathcal{D}} \mathcal{L}(\text{score}(i,j), r_{ij}) + \lambda_1(\|\mathbf{w}\|_2^2 + ||\mathbf{a}||_2^2 + ||\mathbf{b}||_2^2) + \lambda_2\left(\|\mathbf{U}\|_2^2 + \|\mathbf{V}\|_2^2\right) + \frac{\lambda_{rr}}{\text{const} * |\mathcal{U}|} \sum_{(i,j)\in\mathcal{U}} \mathcal{L}\left(\text{score}(i,j), v_{\text{ur}}\right),$$

where $\mathcal{U}$ is a sample of unobserved user-item pairs.

Creating RankingFactorizationRecommender

`graphlab.recommender.ranking_factorization_recommender.` `create` *(observation_data, user_id='user_id', item_id='item_id', target=None, user_data=None, item_data=None, num_factors=32, regularization=1e-09, linear_regularization=1e-09, side_data_factorization=True, ranking_regularization=0.25, unobserved_rating_value=None, num_sampled_negative_examples=4, max_iterations=25, sgd_step_size=0, random_seed=0, binary_target=False, solver='auto', verbose=True, **kwargs)*

Create a RankingFactorizationRecommender that learns latent factors for each user and item and uses them to make rating predictions.
Applied to our model as:

```
#from graphlab.recommender import ranking_factorization_recommender
m1 = graphlab.ranking_factorization_recommender.create(
    train_data,user_id='user_id',item_id='book_id', target='rating',item_data=book_data)
```

Parameters:

observation_data : SFrame

The dataset to use for training the model. It must contain a column of user ids and a column of item ids. Each row represents an observed interaction between the user and the item. The (user, item) pairs are stored with the model so that they can later be excluded from recommendations if desired. It can optionally contain a target ratings column. All other columns are interpreted by the underlying model as side features for the observations.

The user id and item id columns must be of type ‘int’ or ‘str’. The target column must be of type ‘int’ or ‘float’.

user_id : string, optional

The name of the column in observation_data that corresponds to the user id.

item_id : string, optional

The name of the column in observation_data that corresponds to the item id.

target : string, optional

The observation_data can optionally contain a column of scores representing ratings given by the users. If present, the name of this column may be specified variables target.

user_data : SFrame, optional
Side information for the users. This SFrame must have a column with the same name as what is specified by the user_id input parameter. user_data can provide any amount of additional user-specific information.

item_data : SFrame, optional
Side information for the items. This SFrame must have a column with the same name as what is specified by the item_id input parameter. item_data can provide any amount of additional item-specific information.

Comparing the models(here)

Compare the prediction (or model-equivalent action) performance of models on a common test dataset.

```
graphlab.compare(dataset, models, **kwargs)
```

Applied to our models as:

```
model_performance = graphlab.compare(test_data,
[Context_based,Collaborative,popularity,Matrix_factorization])
```

Parameters:
dataset : SFrame
The dataset to use for model evaluation.

models : list[models]
List of trained models.

Result of graphlab.compare:
out : list[SFrame]
A list of results where each one is an sframe of evaluation results of the respective model on the given dataset

The evaluation metric is automatically set to ‘precision_recall’, and the evaluation will be based on recommendations that exclude items seen in the training data.