MODULE 4: RECOMMENDATION SYSTEMS

CASE STUDY ACTIVITY TUTORIAL

CASE STUDY 1 – RECOMMENDING MOVIES

CASE STUDY ACTIVITY TUTORIAL

CASE STUDY 1 – RECOMMENDING MOVIES

In this document, we walk through some helpful tips to get you started with building your own Recommendation engine based on the case studies discussed in the Recommendation systems module. In this tutorial, we provide examples and some pseudo-code for the following programming environments: **R**, **Python**. We cover the following:

1. Getting the data
2. Working with the dataset
3. Recommender libraries in R, Python
4. Data partitions (Train, Test)
5. Integrating a Popularity Recommender
6. Integrating a Collaborative Filtering Recommender
7. Integrating an Item-Similarity Recommender
8. Getting Top-K recommendations
9. Evaluation: RMSE
10. Evaluation: Confusion Matrix/Precision-Recall

Getting the Data

For this tutorial, we use the dataset(s) provided by MovieLens. MovieLens has several datasets. You can choose any. For this tutorial, we will use the 100K dataset dataset. This dataset set consists of:

- 100,000 ratings (1-5) from 943 users on 1682 movies.
- Each user has rated at least 20 movies.
- Simple demographic info for the users (age, gender, occupation, zip)

Download the "*u.data*" file. To view this file you can use Microsoft Excel, for example. It has the following tab-separated format: **user id** | **item id** | **rating** | **timestamp**. The timestamps are in Unix seconds since 1/1/1970 UTC, EPOCH format.

Working with the Dataset

The first task is to explore the dataset. You can do so using a programming environment of your choice, e.g. **Python** or **R**.

In R, you can read the data by simply calling the *read.table()* function:

```
data = read.table('u.data')
```

You can rename the column names as desired:

colnames(data) = c("user_id", "item_id", "rating", "timestamp")

Since we don't need the timestamps, we can drop them:

data = data[, -which(names(data) %in% c("timestamp"))]

You can look at the data properties by using:

str(data)

summary(data)

Plot a histogram of the data:

hist(data$rating)

In Python, you can convert the data to a Pandas dataframe to organize the dataset. For plotting in Python, you can use MatplotLib. You can do all the operations above (described for R), in Python using Pandas in the following way:

```
import matplotlib as mpl
mpl.use('TkAgg')
from matplotlib import pyplot as plt
import pandas as pd
import numpy as np

col_names = ["user_id", "item_id", "rating", "timestamp"]
data = pd.read_table("u.data", names=col_names)
data = data.drop("timestamp", 1)
data.info()

plt.hist(data["rating"])
plt.show()
```

Data Sparsity:

The dataset sparsity can be calculated as:

$$Sparsity = \frac{Number\ of\ Ratings\ in\ the\ Dataset}{(Number\ of\ Movies/Columns) * (Number\ of\ Users/Rows)} * 100\%$$

In R, you can calculate these quantities as follows:

```
Number_Ratings = nrows(data)
Number_Movies = length(unique(data$item_id))
Number_Users = length(unique(data$user_id))
```

In Python, while using Pandas, can you do the same:

```
Number_Ratings = len(data)
Number_Movies = len(np.unique(data["item_id"]))
Number_Users = len(np.unique(data["user_id"]))
```

Sub-setting the data:

If you want the data to be less sparse, for example, a good way to achieve that is to subset the data where you only select Users/Movies that have at least a certain number of observations in the dataset.

In R, for example, if you wanted to subset the data such that only users with 50 or more ratings remained, you would do the following:

```
data = data[ data$user_id %in%  names(table(data$user_id))[table(data$user_id) > 50] , ]
```

Recommenders

If you want to build your own Recommenders from scratch, you can consult the vast amounts of academic literature available freely. There are also several self-help guides which can be useful, such as these:

- Collaborative Filtering with R;
- How to build a Recommender System;

On the other hand, why build a recommender from scratch when there is a vast array of publicly available Recommenders (in all sorts of programming environments) ready for use? Some examples are:

- RecommenderLab for R;
- Graphlab-Create for Python (has a free license for personal and academic use);
- Apache Spark's Recommendation module;
- Apache Mahout;

For this tutorial, we will reference **RecommenderLab** and **Graphlab-Create**.

Splitting Data Randomly (Train/Test)

A random split can be created in R and Pandas (Python).

In R, you can do the following to create a 70/30 split for Train/Test:

```
library(caTools)
spl = sample.split(data$rating, 0.7)
train = subset(data, spl == TRUE)
test = subset(data, spl == FALSE)
```

In Pandas (Python), using the **SciKit-Learn** library, we can do the same via:

```
import pandas as pd
import numpy as np
from sklearn.cross_validation import train_test_split

# assuming pdf is the pandas dataframe with the data

train, test = train_test_split(pdf, test_size = 0.3)
```

Alternatively, one can use the Recommender libraries (discussed earlier) to create the data splits.

For RecommenderLab in R, the documentation in Section 5.6 provides examples that will allow random data splits.

Graphlab's Sframe objects also have a *random_split()* function which works similarly.

Popularity Recommender

RecommenderLab, provides a popularity recommender out of the box. Section 5.5 of the RecommenderLab guide provides examples and sample code to help do this.

GraphLab-Create also provides a Popularity Recommender. If the dataset is in Pandas, it can easily integrate with GraphLab's Sframe datatype as noted here. Some more information on the Popularity Recommender and its usage is provided on the popularity recommender's online documentation.

Collaborative Filtering

Most recommender libraries will provide an implementation for Collaborative Filtering methods. The RecommenderLab in R and GraphLab in Python both provide implementations of Collaborative Filtering methods, as noted next:

For RecommenderLab, use the "UBCF" (user-based collaborative filtering) to train the model, as noted in the documentation.

For GraphLab, use the "Factorization Recommender".

Often, a regularization parameter is used with these models. The best value for this regularization parameter is chosen using a Validations set. Here is how this can be done:

1. If the Train/Test split has already been performed (as detailed earlier), split the Train set further (75%/25%) in to Train/Validation sets. Now we have three sets: Train, Validation, Test.
2. Train several models, each using a different value of the regularization parameter (usually in the range: (1e-5, 1e-1).
3. Use the Validation set to determine which model results in the lowest RMSE (see Evaluation section below).
4. Use the regularization value that corresponds to the lowest Validation-set RMSE (see Evaluation section below).
5. Finally, with that parameter value fixed, use the trained model to get a final RMSE value on the Test set.
6. It can also help plotting the Validation set RMSE values vs the Regularization parameter values to determine the best one.

Item Similarity Filtering

Several recommender libraries will also provide Item-Item similarity based methods.

For RecommenderLab, use the "IBCF" (item-based collaborative filtering) to train the model.

For GraphLab, use the "Item-Item Similarity Recommender".

Item Similarity recommenders can use the "0/1" ratings model to train the algorithms (where 0 means the item was not rated by the user and 1 means it was). No other information is used. For these types of recommenders, a ranked list of items **recommended** for each user is made available as the output, based on "similar" items. Instead of RMSE, a Precision/Recall metric can be used to evaluate the accuracy of the model (see details in the Evaluation Section below).

Top-K Recommendations

Based on scores assigned to User-Item pairs, each recommender algorithm makes available functions that will provide a sorted list of top-K items most highly recommended for each user (from among those items **not** already rated by the user).

In RecommenderLab, the parameter *type='topNlist'* to the *evaluate()* function will produce such a list.

In GraphLab, the *recommend(K)* function for each type of recommender object will do the same.

Evaluation: RMSE (Root Mean Squared Error)

Once the model is trained on the Training data, and any parameters determined using a Validation set, if required, the model can be used to compute the error (RMSE) on predictions made on the Test data.

RecommenderLab in R uses the *predict()* and *calcPredictionAccuracy()* functions to compute the predictions (based on the trained model) and evaluate RMSE (and MSE and MAE).

Graphlab in Python also has a *predict()* function to get predictions. It provides a suite of functions to evaluate metrics such as rmse (*evaluation.rmse()*, for example).

Evaluation: Precision/Recall, Confusion Matrix

For the top-K recommendations based evaluation, such as in Item Similarity recommenders, we can evaluate using a Confusion Matrix or Precision/Recall values. Specifically,

i. **Precision:** out of K top recommendations, how many of the true best K songs were recommended.
ii. **Recall:** out of the K best recommendations, how many were recommended.

In RecommenderLab, the *getConfusionMatrix(results)*, where *results* is the output of the *evaluate()* function discussed earlier, will provide the True Positives, False Negatives, False Positives and True Negatives matrix from which Precision and Recall can be calculated.

In Graphlab, the following function will also produce the Confusion Matrix: *evaluation.confusion_matrix()*. Also, if comparing models, e.g. Popularity Recommender and Item-Item Similarity Recommender, a precision/recall plot can be generated by using the following function: *recommender.util.compare_models(metric='precion_recall')*. This will produce a Precision/Recall plot (and list of values) for various values of K (the number of recommendations for each user).