MODULE 4: RECOMMENDATION SYSTEMS

CASE STUDY ACTIVITY TUTORIAL

CASE STUDY 2 – RECOMMENDING NEW SONGS BASED ON USER LISTENING HABITS

CASE STUDY ACTIVITY TUTORIAL

CASE STUDY 2 – RECOMMENDING NEW SONGS BASED ON USER LISTENING HABITS

Problem: Recommend New Songs to users based on their listening habits (e.g. Spotify)

1. Subset of about 10k songs
2. 0/1 "ratings": listened/not-listened
3. Over 99% sparse.

- **Dataset**: Million Song dataset (http://labrosa.ee.columbia.edu/millionsong/)
  - Million Song Dataset has several datasets. You can choose any. For this example we will use the 10K dataset (~1.8Gb) - http://static.echonest.com/millionsongsubset_full.tar.gz
  - A more manageable extract of this dataset is also available at: http://s3.amazonaws.com/dato-datasets/millionsong/10000.txt
  - Save this file as "**songs.txt**"
  - This data set consists of:
    - 10,000 songs
    - The file has the following tab-separated columns:
      user_id | song_id | listen_count
    - Additional metadata about the songs is available here: http://s3.amazonaws.com/dato-datasets/millionsong/song_data.csv
      - This file contains the following columns: **song_id, title, release, artist_name, year**
      - Save this file as "**song_metadata.txt**"

- **Read/View 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('songs.txt')`
    - You can rename the column names as desired:
      `colnames(data) = c("user_id", "song_id", "listen_count")`
    - Since we don't need the listen_count, we can drop it:
      `data = data[ , -which(names(data) %in% c("listen_count"))]`
    - Additionally, if you want to associate song names with the IDs, you can do so by first reading the song metadata:
      `song_metadata = read.csv("song_metadata.csv")`
    - Now you can merge the two datasets:
      `data = merge(aata, song_metadata, by="song_id")`
      `data = droplevels(data)`

- Now you can look at the data properties by using:
  ```
  str(data)
  summary(data)
  ```

- In Python, you can convert the data to a pandas dataframe to organize the dataset and merge with the metadata (http://pandas.pydata.org/)

- The dataset sparsity can be calculated as:
  - Sparsity = <Number of rows in the Dataset> / (Number of songs x Number of Users) * 100%

- **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/Songs 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:
    - http://www.salemmarafi.com/code/collaborative-filtering-r/ ,
    - http://blogs.gartner.com/martin-kihn/how-to-build-a-recommender-system-in-python/

  - 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 in R (https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf),
    - Graphlab-Create for Python (has a free license for persona and academic use)(https://dato.com/products/create/docs/graphlab.toolkits.recommender.html),
    - Apache Spark's Recommendation module (https://spark.apache.org/docs/1.4.0/api/python/pyspark.mllib.html#module-pyspark.mllib.recommendation),
    - Apache Mahout (https://mahout.apache.org/users/recommender/userbased-5-minutes.html)

- **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, using the SciKit-Learn library:

```
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 (https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf)
  - Graphlab's Sframe objects also have a random_split function which works similarly (https://dato.com/products/create/docs/generated/graphlab.SFrame.random_split.html)

- **Popularity Recommender:**
  - The **RecommenderLab** in R, for example, provides a popularity recommender out of the box. Section 5.5 of the RecommenderLab guide provides examples and sample code to help do this: https://cran.r-project.org/web/packages/recommenderlab/vignettes/recommenderlab.pdf

  - **GraphLab**-Create also provides a Popularity Recommender (in python). If the dataset is in Pandas, it can easily integrate with GraphLab's Sframe datatype as noted here: https://dato.com/products/create/docs/generated/graphlab.SFrame.html. Some more information on the Popularity Recommender and it's usage is provided here: https://dato.com/products/create/docs/generated/graphlab.recommender.popularity_recommender.PopularityRecommender.html

- **Item Similarity Filtering:**
  - Recommender libraries will also provide Item-Item similarity based methods.
    - In **RecommenderLab**, use the "IBCF" (item-based collaborative filtering) to train the model.
    - In **GraphLab**, use the "Item-Item Similarity Recommender" https://dato.com/products/create/docs/generated/graphlab.recommender.item_similarity_recommender.ItemSimilarityRecommender.html

    - 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.
    - A Ranked List of items recommended for each user is provided, 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 will do the same.

- **Precision/Recall, Confusion Matrix:**

  - For the top-K recommendations based evaluation, such as in Item Similarity Recommender, 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 **evalute()** 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()

  - In GraphLab, 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).