STA141C Final Report
Yuji Mori
June 2017

Building Recommenders with Steam Video Game Data

Introduction

Steam is a digital distribution platform with a dominating presence in the video game market. There are over 40 million active users who purchase and play games through this software. In e-markets such as this one, **recommender systems** are a staple feature to expose users to new products and boost sales. The personalization of these recommendations can be greatly attributed to advancements in machine learning, so in this report, I will be demonstrating several of these algorithms/procedures using data from Steam user/game interactions.

Ideally, a recommendation engine would be supplied as much information as possible about both the users and the products. But in this case, user profile data (such as location, gender, age) is naturally unavailable. Due to the nature of my data, all of these methods are considered **Collaborative Filtering (CF)** algorithms. CF is a useful alternative strategy that does not depend on the creation of explicitly defined user profiles; instead, useful recommendations can be generated from a collection of past user behavior.

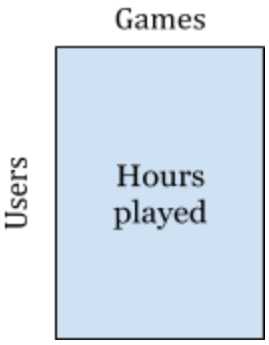
Recommender systems rely on different types of input. The most convenient is high quality explicit feedback, which includes explicit input by users regarding their interest in an item (for example, Netflix collects star ratings to indicate how much they enjoy a movie). However in my case, the dataset only contains **implicit feedback**, which indirectly reflects user opinion through observing user behavior (i.e. overall playtime of each game). When working with implicit feedback, one must take many precautions with their model. The relevant details will be provided in a proceeding section.

Data and Data Pre-processing

The primary dataset is a very straightforward record in the following form:

Steam User ID	Game	Hours Played

Each row corresponds a unique user/game interaction. However, this will be pivoted into a sparse matrix form that is ubiquitously used by most CF recommenders, as shown on the right. Furthermore, noisy interactions (users who purchased too few games) will be removed, as these users can neither offer insightful improvements to our model, nor can the model make useful recommendations for them. After all processing steps, I ended with a matrix of shape (1312, 3429). **The goal of CF recommenders is to predict the unobserved entries in the user/item matrix (cells that are currently zero).**



STA141C Final Report
Yuji Mori
June 2017

TRAINING AND TESTING SETS.

I. It is typical to split the data for model evaluation purposes, but for recommender systems, the process is rather different from the usual 80-20 rule; instead of splitting by row, it is necessary to randomly mask some percentage of non-zero cells from our User/Item matrix, and train the model using an "incomplete" matrix. Then, the performance of the model is tested against the matrix containing the entire set of data to see if successful recommendations were made.

II. I implemented a novel method of model evaluation by collecting the most recent data from the **Steam Web API**. While my updated observations were collected in June 2017, the original data (used to train the model) is not precisely dated, though the measurements were taken prior to March 2017. Using this 3+ month gap, I created an identical User/Item matrix with new real ratings, instead of masking known ratings like before.

I will frequently refer to these as procedure (I) and procedure (II) in the rest of the report.

Method 1: Item-Item Collaborative Filtering

Two games (i) and (j) can be thought of as two vectors in the (m)-dimensional user-space. The similarity between them is measured by computing the cosine of the angle between these two vectors, as shown below:

$$sim_{(i,j)} = cos_{(\mathbf{i},\mathbf{j})} = \frac{\mathbf{i} \cdot \mathbf{j}}{\|\mathbf{i}\| \cdot \|\mathbf{j}\|}$$

Intuitively, overlapping vectors with cosine value of 1 indicates total similarity (or per user, across all games, there is same rating), while cosine of 0 would indicate no similarity. Side note: some papers have proposed using correlation-based similarity or adjusted cosine similarity. Either way, the purpose is to construct an item-item similarity matrix of all (N) items, which is then used to predict a rating (p) that a user (u) would give to some item (i):

$$p(u,i) = \frac{\Sigma_{N \in Ksimilar}(s_{i,N} \cdot R_{u,N})}{\Sigma_{N \in Ksimilar}(|s_{i,N}|)}$$

In many cases, it may be useful to normalize the rating and deal with a binary classification problem (like/dislike). I took a slightly different approach by implementing my own accuracy calculation: from the predictions, I analyzed the top 10 predictions for each user, and assessed whether the user actually purchased at least **one** of the recommended items. Using the training set from procedure (I), I obtained an accuracy rate of about 0.560, and from procedure (II), it was much lower at 0.37. These scores can be compared to the baseline probability (chance of a random hit), which is equal to: $\frac{(10\ random\ items)}{(total\ items\ in\ dataset)} = 0.02916$

Although both accuracy readings are much higher than the baseline, procedure (II) is substantially worse performing than (I). I suspect that there were too many issues with combining data from two

different sources; even with extensive text processing, I could not retrieve a small percentage of game titles that were found in the original (non-API) dataset.
One problem here is that users do not inherently purchase games at random; a better metric would be a comparison illustrating how the CF recommender beats out an algorithm that simply supplies the top 10 most popular games. The **average AUC** (area under the ROC curve) can give the tradeoff between true positives (recommendations that were purchased) and false positives. Though we expect these values to be low comparing the top recommended and most popular (There is very little chance that a user actually went and purchased precisely those 10 games), it is the difference in these values that we are interested in. A result of (0.217, 0.154) was obtained, with our item-item CF recommender model being higher. Closely related, another known metric for implicit model evaluation is precision-recall, which could possibly outperform the AUC measurements.

Final comments:
Despite not using any information about user profiles, the item-item CF model based on cosine similarity performed rather well. Next, the model performance will be compared to another CF method.

Method 2: Matrix Factorization (Latent Factor Model)

Matrix factorization assumes there are some underlying (latent) features that can be used to describe the user/item interaction as a way of predicting the unknown values inside the matrix. There are many forms of matrix factorization that have been applied to CF recommenders, but for implicit feedback data, there has been much success using Alternating Least Squares (ALS) with regularization. A unique component of the model I am using is the binarization of the implicit ratings ($p_{u,i}$), while retaining the numeric component as a confidence measure ($c_{u,i}$). Specifically, this is the cost function to be minimized:

$$\min_{x_*,y_*} \sum_{u,i} c_{u,i}(p_{u,i} - x_u^T y_i)^2 + \lambda(\sum_u \|x_u\|^2 + \sum_i \|y_i\|^2)$$

The "Alternating" part comes into play by iteratively adjusting these two vectors until convergence:

$$x_u = (Y^T C^u Y + \lambda I)^{-1} Y^T C^u p(u)$$
$$y_i = (X^T X + X^T (C^i - I) X + \lambda I)^{-1} X^T C^i p(i)$$

I encountered some problems tuning the hyperparameters, but below I present my best results. Originally I wanted to show the MSE from each iteration of minimizing the loss function, but it did not work very well with my Python implementation of ALS. Lambda was set to 0.01, and I used 200 factors to train the ALS model.
This model performed slightly better than the item-item similarity model, which is consistent with many other published findings. The accuracy using procedure (I) was 0.596, and 0.492 for procedure (II). In both cases, the ALS model outperformed the item-item model, most likely because this method is essentially considers both user-user and item-item relationships to construct predictions.

STA141C Final Report
Yuji Mori
June 2017

Method 3: TF-IDF on Game Descriptions

The previous two methods were examples of collaborative filtering, but next I will demonstrate a recommendation-generating procedure that does not depend on any user/item interactions. Using a new dataset by Craig Kelly, I have used game descriptions from the Steam online store (one or two sentences describing each game) to build a similarity matrix; each game is represented as a feature vector of words, and the cosine between each pair of games is calculated. Once again, this does not use any sort of user/item data, and the calculations are made solely based on product characteristics.
This implementation is applicable to many more data sources, but the results are not necessarily personalized. For example, the following is the result for games similar to the game, “Half-Life 2”:

Cosine Similarity	Game
0.089030089716296223	'Half-Life Source'
0.073918511783881646	‘Half-Life 2 Episode One'
0.057412315995251123	'Half-Life'
0.055727621947850421	'Half-Life Deathmatch Source'
0.053535523479960559	‘Black Mesa'
0.046257237480748822	'Half-Life 2 Episode Two'
0.040617023140805418	'Half-Life Opposing Force'
0.039560043491573077	'Day of Defeat Source'
0.035230375561999004	'Half-Life Blue Shift'
0.029697592157274352	'Left 4 Dead’

This is a known limitation of any kind of item-based recommender, known as the “Harry Potter Problem”: These are indeed the most similar items, but the user is most likely aware of, or have already purchased, these titles, making the recommendations less useful. It is even possible that when using Procedure (I) to mask existing user/item interactions, the accuracy is inflated due to this problem.

Concluding Remarks

Though implicit data is difficult to work with, there are a variety of methods that allow for reasonable recommendations. My analysis indicates that matrix factorization with ALS is the most reliable method for generating top-10 lists, though the item-item recommender has shown

competitive results. Though overall, the purpose of this report is not necessarily to assess which model performs better, but rather to determine which attributes of the data cause problems for each model, and how those problems can be overcome by taking building an algorithm that is cautionary and effective at the same time. This report has shown that model validation procedures also greatly affect the model performance; "masking" user/item interactions is by far the most popular method of testing recommender models, but I also attempted to use temporal changes in the dataset to simulate a more "real" outcome, which in turn greatly reduced by model accuracy.

References

Papers and Online Sources:

A Gentle Introduction to Recommender Systems with Implicit Feedback. Jesse Steinweg-Woods, Ph.D.
https://jessesw.com/Rec-System/
A Simple Content-Based Recommendation Engine in Python. Chris Clark.
http://blog.untrod.com/2016/06/simple-similar-products-recommendation-engine-in-python.html
Collaborative Filtering for Implicit Feedback Datasets. Yifan Hu, Yehuda Koren, Chris Volinsky.
http://yifanhu.net/PUB/cf.pdf
Item-based Collaborative Filtering Recommendation Algorithms. Badrul Sarwar, George Karypis, Joseph Konstan, and John Riedl.
http://files.grouplens.org/papers/www10_sarwar.pdf

Data:

200k Steam User Interactions by Tamber:
https://www.kaggle.com/tamber/steam-video-games
Steam Data by Craig Kelly:
https://github.com/CraigKelly/steam-data

Code / Python Modules:

Implicit by Ben Frederickson:
https://github.com/benfred/implicit
Content-based Recommendation Engine by Chris Clark:
https://github.com/groveco/content-engine/blob/master/readme.md
SteamAPI by smiley:
https://github.com/smiley/steamapi
steam by ValvePython:
https://github.com/ValvePython/steam