CS51 Final Project: TOONZ

Rachael Smith ~ Anna Zhong ~ Alex Lehman

OVERVIEW

We implemented an item-item collaborative filtering algorithm to predict how much a user will like a song that they have not listened to. Our prediction is given in the form of a projected play count. Our collaborative-filtering mechanism is adapted from the mechanism laid out by the course materials provided by the Harvard University CS109 Course Staff[1]. It first predicts a baseline value for the user's number of listens predicted for a given song. The baseline estimate is calculated via the formula:

$$B_{us} = m + (m_u - m) + (m_s - m)$$

where m is the average number of listens of all the songs over all the users in the dataset, m_u is the average number of listens for the user across all songs, and m_s represents the average number of listens the particular song has across all users. The second and third terms of the expression are meant to count for user biases and item-specific biases in the data.
Then to implement our item-item collaborative filter we use the following model:

$$R_{ut} = B_{ut} + \frac{\sum_{i \epsilon N^k(t)} s_{jt} (R_{uj} - B_{uj})}{\sum_{i \epsilon N^k(t)} s_{jt}}$$

Here, R_{ut} is the predicted number of listens of song *t* for user *u*. $N^k(t)$ represents a function which gets the k most similar songs to a song t among the songs user u has listened to. s_{jt} represents the similarity between two songs $j$and t. ($R_{uj} - B_{uj}$ is the difference between the number of times u actually listened to song j and the baseline prediction of how many times u would listen to j.

Our algorithm outputs both R_{ut} and m_u, so one can tell how much a user will like a given song by comparing the average number of times the user will listen to any given track and the predicted number of times the user will listen to the particular track, *t.* In order to implement this algorithm, we first build up a database of song-song similarities using a pearson similarity function and user play count data, which was then used to extract values of s_{jt}.

PLANNING

Our project was first conceived from a suggestion from our TF: implement a collaborative filtering algorithm to build an item recommender. Rachael, an avid WHRBie[2], thought it would be especially fun and interesting to create a music recommender, and in that auspicious moment, BAM! TOONZ was born. We then set out upon a Google search, trekking deep into the depths of the World Wide Web to unearth all the information we could find on this so-called "collaborative filtering algorithm".

[1] Accessed at http://cs109.org/
[2] 'WHRBie' = Member of Harvard College Radio

Through a synthesis of this data we formulated our **draft spec** in which we boldly declared that we shall implement an item-item based CF. Should time allow, we hoped to also a user-user CF to integrate into a hybrid CF. In said draft spec we outlined three major milestones of our implementation: building a database of song similarities, generating global user ratings, and implementing a collaborative filtering mechanism. Foolhardily, we did so before we had chosen an API to use as our data source and a particular CF mechanism we wanted to implement. Thus our original plan, though a valiant effort, was ultimately misguided in that it failed to account for how we would need to manipulate the API data to fit our needs, and how we would need to manipulate the CF mechanism to fit the particular data.

Our **final spec** reflects these changes. Our milestones remained the same, but our map for implementing them was altered significantly. Initially we were planning on implementing a CF which would predict user ratings of songs. However, our data only provided us with user's song play counts, so we would need to alter our CF to predict user's song play counts instead.

This reconceptualization of our project was definitely a set-back, but we bounced back heartily. Our milestones were met mostly with success. We were able to complete all of them (hurrah!). However, they took longer to code than we were anticipating. In planning our timeline we underestimated how long it would take for us to adjust to working in Python and we failed to account for how long testing this algorithm would take. Testing proved to be a much larger undertaking then we had imagined. Once complete, we had no time left to implement our "hopeful" features ☹ .

DESIGN AND IMPLEMENTATION

The biggest difficulty we faced in the design and implementation phase was adjusting to Python, a language that was fairly new for two of us. Although, it feels funny to say that adjusting from OCaml to Python is really 'difficult'. How refreshing it was to be working with a dynamically typed language once again!! We chose Python because we needed an easy way to handle a large data structure, and the pandas library allowed us to do just that. The data we used is a taste-profile subset of EchoNest's million songs API. We read this taste-profile dataset into a DataFrame, a structure provided by pandas.

Manipulating the data

The DataFrame essentially provided a representation of a table containing columns representing unique user ids, unique song ids, and user's song play counts. We needed to manipulate this data to add two columns to the table representing a user's average play count across all the songs they listened to, and the average number of times a given song was played across all users. Pandas made this incredibly easy to code, however we ran into trouble implementing this due to the large size of our dataset. When we first started running operations on our DataFrame, they took huge amounts of time because we would try and run them on the whole set.

We tried to create a smaller subset of the data to use for testing and debugging by removing all songs for which had fewer than 50 listens. The result of this reduction was still way too big to work with, at least to start out with (over 100,000 items remained). We then tried a data frame of less than 50,000 rows (which amounts to a few thousand users), which was able to run in reasonable time. However, this did not give us many pairs with significant similarity between songs due to the data being too sparse, since these few thousand listeners were spread among a million songs. In the end, what we decided to do was to manually create a very small dataset with 3 users and about 20 distinct songs (with some shared songs) to use to implement our similarities database functions. This was very helpful in that since

we knew the data we were working with, we were able to predict expected values of functions for testing purposes. However, the extremely small size it made it difficult to tell how accurate our similarities calculations are at this point in our implementation and this proved a drawback later on.

Similarities database

Implementing the similarities database went considerably more smoothly. We did find that we needed to break this process up into smaller steps then we were originally envisioning. However, as we spent more time getting familiar with the pandas library, we also found that pandas made it much easier to implement many things. For instance, we imagined that the first step in building the database of song-song similarities would be to build a somewhat complex data structure containing a song-song pair and list of their shared listeners. However, it turns out that the shared listeners of a given song-song pair can be easily extracted using native pandas functions. One simply indexes the DataFrame to extract the sub-frame consisting of all the listeners of song A, and then the does the same for song B, and then intersects the results, voila! Like magic, pandas turned what we thought would be a complex process into a three lines of code. Now, if only we hadn't wasted so much time trying to do it the other way…

However, it turns out that populating the similarities database for each pair of songs was the limiting factor in our code's run time. This is because for n given items, the number of pairs formed between all items is n*(n-1)/2, which means that the algorithm has $O(n^2)$. Therefore, even for small amounts of data added to our dataset, the worst case run time would increase dramatically. No wonder our program couldn't get past this stage when we ran it on our 100,000-row dataset!

When calculating similarities between songs, we also needed to account for the possibility that some songs did not have any common listeners in our data set. To do this, we shrunk our similarity value slightly based on a value that we implemented so that it can be adjusted based on the makeup of the data set being analyzed. This was done in our reg_sim function.

K-nearest neighbors

We then needed to find a way to produce a list of "neighbors" given a certain song as a base for song recommendations. In order to this, we applied our song similarity function between our one base song and each of the remaining songs in the data set of interest. This produced a list of the songs along with their similarities to our base song. In order to get the top 5 (or any number you desire) that are most similar to our base song, we sorted this list of similarities in decreasing order and returned the top 5. It was conceptually difficult to figure out how we were to make sure to get the top *most* similar songs. Because the database is so big, we were weary of applying the similarity function to every set of songs in our data set. This is where our handy-dandy similarities database came in handy! Since we had already calculated and stored them, all we had to do was a simple lookup.

Recommending songs

Final implementation difficulties came in how to recommend songs to the user that he/she had *not* already listened to. In order to do this, we had to set aside a list of the songs that had been listened to by the user and check to make sure the ones we recommended were not in this list. Finally, using this function we were able to recommend songs to users and extract the song names using just the id from the API. This, however, also proved difficult because the Echonest API that complemented the dataset we were using had been updated, while the dataset hadn't changed. Therefore, when calling for the titles of songs in our dataset, some returned errors since they no longer existed in the Echonest API. We fixed this by handling the exception to just print the song id with a note about the problem. This is acceptable in the scope of the final project, but if we were to implement our recommendation engine

for real users, we would need to obtain a different dataset with updated song values within the API because it wouldn't be very helpful if people couldn't know the names of the songs they were being recommended!

Testing

The last part that was hard to figure out was how to test if our recommender was actually working! We needed a way to find out if our predictions of what people would like were good ones or not. In order to do this, we decided to compare results from our pred_plays function on the actual plays for songs that a user had *already* listened to. That way, we could see how close our predicted play counts for a given user were compared to the actual numbers! For our small dataset with 3 users and 20 songs, the results came out exactly as predicted, with no margin of error (i.e. for our test user, all the songs that were being tested had matching predicted play counts and actual play counts). Because our data was skewed, we manually created a slightly larger dataset of 10 users, again with songs that would give us significant results. They were more interesting to analyze. Here are the results for one test user:

For **user** 5a905f000fc1ff3df7ca807d57edb608863db05d | **avg** 3.90909090909:

Song	Predicted Play Count	Actual User Play Count	Average Play Count
Roral	9.73390842734	20	11
The Best of Times	5.99997499547	16	6
Auto-Dub	5.73390842734	7	7

As you can see, the predicted play count is lower than the actual play count, with results deviating more the higher the actual play count is. The reason for this is because we compared songs from a subset (called "songz" in our code) that was taken from a *sorted* list of songs that a user had listened to. Therefore, the actual play counts are higher because these are the test user's top three choices, representing the largest positive deviations away from his or her average play count of 3.9090. Because the predicted play counts were constructed from information taken from the K nearest neighbors, they should fall closer to the user's mean than the actual play counts do. Taking into account the song's overall average play count helps a little bit here because we can adjust the predicted play count to reflect a good song, but it does not counteract the effect of us looking only at the higher end of the user's actual play count data. Nevertheless, the order in which these recommendations appear reflect the true order (all predicted play counts and actual play counts are in descending order), meaning that the first song recommended to any user with Toonz would indeed be the song that the user would listen to most.

Reflections

One thing we would do differently, were we to do the project again, would definitely be planning. In our specs, we outlined our goals and specifics about the functionality we wished to implement, but we didn't create a concrete timeline for accomplishing those goals. Creating a timeline forces one to think deeply about what each step of the process requires and consider possible problems which might arise in order to estimate how much time each step will take. Thus, creating a timeline inevitably enables more realistic and more accurate estimations about how much time different aspects of a project will take. If we had set out a timeline from the get go, we may have been able to implement more functionality then we ended up being able to.

Relatedly, we found that beginning to plan the project without finishing our research was a not-so-good idea. We began to plan our project around the idea that we would be using a dataset that had information about user song-ratings. However, the dataset we ended up using did not have this information, so we had to rethink many aspects of our project. If we had researched and selected our API first, we would not have wasted this time.

On the other hand, a choice we were quite pleased with was decided to do the project in Python. Learning a new language is always both a challenging and rewarding experience. After struggling to master Objective-C for their CS50 project last semester, Anna and Rachael enjoyed learning Python, as it was a much easier language to pick up on. A downside to working in a new language is that we often started out trying to implement things in a way that was unnecessarily complex due to our lack of familiarity with the Python/pandas libraries. On the other hand, this can kind of be viewed as an upside: the language we chose was able to help us simplify our implementations immensely.

As mentioned earlier, our results are reasonably accurate because we manipulated our very small dataset to produce desirable similarities between certain songs that we could test on. Although this proved our algorithm to be functional, we can only say that it is accurate for data that has very little distribution. The usefulness of Toonz on an item-based CF algorithm cannot be fully evaluated until we run it on larger, more relevant datasets that currently take days to run on our computers. Ideally, we would've liked to implement MPI or map-reduce algorithms to split the work amongst several processors to do it in very little time. In theory, we know that other more effective recommendation algorithms exist, such as Bayesian model predictions. If we had more time, we would've liked to implement the Bayesian model as well and compare results from it with those from item-based CF.

Advice to Future Students

One thing we would definitely recommend to future students is to split up the work, but not to split up ALL the work. While all three of us each had distinct parts of the project we were primarily responsible for (Anna- the video, Rachael- the write up, Alex- testing the code), we did a lot of the coding together, especially at the beginning. Although it may be harder to coordinate (trying to find time for any group 3+ Harvard students to meet is always a challenge) this strategy has several advantages as opposed to splitting up the code into individual tasks. For one, it keeps everyone in the group on the same page for all parts of the project. By working collaboratively, we each had a much deeper understanding of how all parts of the code work together as a cohesive whole. Secondly, the beginning of a large coding project is often the most conceptually difficult stage, so it is better to have three minds rather than one when getting started. For our group in particular this was a successful strategy because Alex was the only one of us with previous experience working in Python. By working together on the code, Anna and Rachael were able to learn a great deal from her expertise.

The most amount of time spent on this project was running our code on datasets that were too big and which could never be completely processed. We didn't know this until we had spent almost 2 days waiting for our code to finish running, before giving up and testing out smaller datasets on a trial-and-error basis. This was trial-and-error because we weren't sure of how to balance a smaller dataset (which runs much quicker) with having enough information for significant similarities to be made between songs (and therefore more accurate recommendations). In the end, as mentioned above, we just created our own specific datasets with results that we could analyze. Obviously, these problems weren't

obvious to us until very later on in the project, but being aware of them would've helped save a lot of time wasted on trying useless datasets.

It has been said before and it is sort of self-apparent, but we'd also like to recommend to future students that they get started early on their project! Perhaps the most important (and most over-looked) factor in a successful project is taking the planning stages very seriously. Creating a hard timeline is likely very helpful, and it something we wish we had done for our project. When planning, always overestimate how long each part will take, because unexpected challenges always come up and there is always something that you forgot to account for!