Final Report

By: Matthew Wishoff, Thomas Tse, and David Chau

Date: 12/10/2016

CSC 849 Search Engines

Table of Contents

Approach

Throughout this project there has been a lot of trial and error. Initially we wanted to make an Android application that would recommend video games. We were drawn to making an Android application, because it's mobile and easy to use and goes with you everywhere. However we ran into trouble getting Apache Solr to interface with the Android application. Our idea was to have Apache Solr run on Matt's Raspberry pi, and use that as a server for our Android application to use. However, Comcast apparently does not like it when you set up servers in your house, and restricts the flow of data through the router. I'm sure there might have been a work around to get it working, however we did not think that we had the time to take a chance on getting router magic to work. Therefore we decided to pivot, and make a web application instead. We all have experience making web applications, so we thought this would be a great way to pivot the same idea to a different delivery device.

Now that we're working with a website instead of an Android application, we thought we would be able to host Apache Solr on the same server the website is being hosted on. This idea worked! However we were now having yet another problem with Apache Solr. When giving Apache Solr the HTML documents to index for us, we found that it wasn't indexing all of the information we wanted from the webpage. Given that we were having so much trouble getting Apache Solr to work we didn't want to risk trying to get Indri to work. Therefore we decided to go through the HTML documents and parse it ourselves, adding all the information we needed to our index. When we index the information it does have some limitations. For example it only works on Steam Community web pages, so it doesn't index general webpages and only the type of page we're getting our data from. This works well for this class since the scope of the project has been refined to just a video game recommendation engine.

Now our approach has been refined to what we are currently doing. We are making a website as the delivery device for our video game recommendation system. We are indexing the data which we collected ourselves. We really liked doing it this way, because it cuts down the wall that is Indri, and Solr that is more of an obstacle to us rather than a help. We feel that if these services were better documented, or had more resources provided for people who need help they might be better.

Collecting Our Data

We wrote a web scraping script that given the name of the game, we go to the page and scrape the Steam Community page for its HTML. To get all the game names that we wanted to use in our data set, we found a user with ~2000 games on their Steam account. We were able to use the Steam API to query this user's account to get the list of games that he/she owns. With this data we are able to run it through our web scraping script that retrieves the web page for the reviews, once we have the HTML scraped we save it to a file saving it for later when we start indexing. Once we've successfully gone through all the user's games we run those HTML files through our indexer, scraping the HTML page for the game id, reviews, and genres. We make sure to save all this data into a new entry in our index each time we read in a new HTML file. While we are building the index of games we simultaneously build our index of genres. This allows us to go through our data set in O(n) time, so that it is a much faster process than going through it twice or more times.

```
"genre_groupings" :
{
        "indie":["war for the overworld", "edge",  ...
        "strategy":["war for the overworld", "spectromancer",  ...
        "adventure":["the gallery  episode 1 call of the starseed", ...
        "action":["edge", "unreal gold", "counterstrike,  ...
        .
        .
        .
},
```

Figure 1 – Example of how we group games into genres

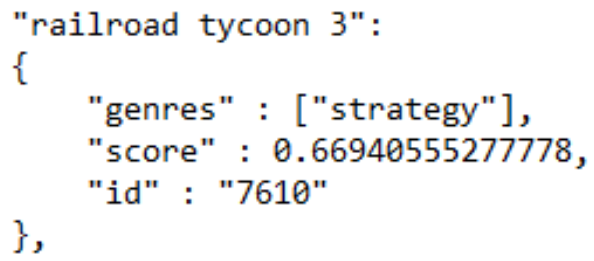

Figure 2 – Example of how we index a game into our main index

Since we have the pipes working to take a new game data, we found it useful to add another piece of functionality to our project which allow us to increase our data set at will. We added a field to our website where given the ID of a game on Steam, it will take that game's web page read in the HTML for its data, and append it to both our genre groupings index, as well as our main game index. Doing this allows us to continually increase our data set in size. We believe that with a few modifications we can change this function to accept a game name instead of a game ID. We chose to do it by game ID because it was easier to implement this way, and it wasn't really in the scope of this class.

Other data that we decided to collect is information about Steam's user base. Using the same kind of call we used to collect a list of games from 1 user, we use the same call to collect information about 8000 users. We simply incremented the SteamIDs by 1 for 8000 times to collect the games for each user. Some Steam accounts have been deleted and others had no games on them. So after this long API query process we had about ~3600 users in our data set, each of them having a list of games linked to their steam account. With this information we were able to write a correlation algorithm. We went through the entire user list, and through the each of the user's games. When two games match the same genre, we add them as a pair to a hash. When that same pairing happens again for another user, we increment that hash entry by 1. This allows us to keep a running count for how many people own the pair of games. We have a threshold of approximately one third of our data set. When the threshold is met we can infer that there is a correlation between these two games. So when someone searches for "Counter Strike Source" and if there is a game with a high correlation with it, that game will also

be recommended to the user, along with the other games we decide to recommend based on sentiment score. Figure 3 is the algorithm which we used to make the correlation hash.

```
foreach($all_user_game_list_json as $obj){
        echo "Vrooom\n";
        foreach($obj as $response => $data){
                //echo $data->game_count;
                echo " Beep beep\n";
                $num_games = $data->game_count;
                $games = $data->games;
                for($i = 0; $i < $num_games; $i++ ){
                        $game1 = parse_key_str($games[$i]->name);
                        $game1_data = $indexer->get_game_data($game1);

                        if($game1_data)
                                for($j = $i + 1; $j < $num_games; $j++){
                                        $game2 = parse_key_str($games[$j]->name);
                                        $game2_data = $indexer->get_game_data($game2);

                                        if($game2_data){
                                                $count = 0;
                                                foreach($game1_data["genres"] as $genre){
                                                        echo "VROOMM VROOOM\n";
                                                        if($genre !== "freetoplay" && $indexer->is_game_in_genre($game2, $genre)){
                                                                $count++;
                                                                break;
                                                        }
                                                }
                                                if($count > 0){
                                                        $key = (strcmp($game1, $game2) < 0) ? $game1 .'-'. $game2 : $game2 .'-'. $game1;

                                                        if(array_key_exists($key, $correlation_hash))
                                                                $correlation_hash[$key] += $count;
                                                        else
                                                                $correlation_hash[$key] = $count;
                                                }
                                        }

                                }
                }
                echo "Checkered Flag!" . "\n";
        }
}
$stream = fopen("correlation.json", "w");
$result = fwrite($stream, json_encode($correlation_hash));
```

Figure 3

Scoring

Initially we thought it would be a good idea to score the games based on positive words (Figure 4), and negative words (Figure 5) in the review. A review with more positive words would obviously be a positive review. Likewise, a review with more negative words would obviously be a negative review. We could sum the scores of the reviews, and the games with lots of positive reviews would be recommended first. However, there was an issue with doing it this way. We found that people use positive words negatively in there reviews most of the time. For example: "I wish this game was better". Better is in the list of positive words, but the sentiment of the sentence is very negative. However If we are just looking at the list of words, this review would get a net score of positive 1. We soon realized that this short way of trying to evaluate reviews was not going to work out as expected with good results.

We started searching for a new way to evaluate our games without changing the kind of query we are doing. Matt talked to his friend, Paul Klein, and he was using a Microsoft Text Analysis API to evaluate customer reviews for positive and negative sentiment. Immediately we knew that this approach is what we needed in our project. The Microsoft Text Analysis API scores the sentiment of the text out of 100%. Below 50% is considered negative sentiment, and above 50% is positive sentiment. What this will allow us to do is score the games based on what people actually said about them in the proper context. Once we score a review we can add that score to the other scores we already have on that game, and then once we have totaled all the review scores simply take the average. This gives us an average score for the game, based on what people actually thought and said about the game. One of the hang ups of when using this API is that it doesn't accept text blocks over 10,240 bytes. So, some very long reviews did not make it into our dataset. Other than that, this API provided us a very clear, and clean way to score the games before a query even happens. Since getting the results from our hashes is in O(1) time it is very good for our recommendation engine.

PositiveWords:
a+
abound
abounds
abundance
abundant
accessable
accessible
acclaim
acclaimed
acclamation
accolade
accolades
accommodative
accomodative
accomplish
accomplished
accomplishment
accomplishments
accurate
accurately
achievable
achievement
achievements
achievible
acumen
adaptable
adaptive
adequate
adjustable
admirable
admirably
admiration

Figure 4 – Subset of positive words

Negative words:
2-faced
2-faces
abnormal
abolish
abominable
abominably
abominate
abomination
abort
aborted
aborts
abrade
abrasive
abrupt
abruptly
abscond
absence
absent-minded
absentee
absurd
absurdity
absurdly
absurdness
abuse
abused
abuses
abusive
abysmal
abysmally
abyss
accidental
accost
accursed

Figure 5 – Subset of negative words

Query Feedback

Looking back at some of the lectures we've had in class, we remembered that one of them included user feedback. We also remembered that users do not like giving feedback by filling out forms and such. Therefore we don't give the user a choice in the matter, and sneakily take feedback from their query results. When we give them the search results, each result is a link to the Steam page where you can buy the game when logged into Steam. When a person clicks a link displayed in our results, we increment a counter associated with the game ID of that result. We also decrement the counters of the results that were not clicked on. By decrementing the counters of other results, we eventually discover the games that are never clicked on because those games will have a really negative score. In contrast, the games that are clicked on very often will have a higher score. Games that are just added to our data set, such as new games come in with a score of 0. If we just incremented games that were clicked on, these games will have a very high scores, but games that are never clicked on will just have a score of 0. Since new games are added in with a score of 0 and that correlates to a game that is never been clicked on, we cannot do it this way. Therefore, we need to decrement to show that a game has been in our dataset for a long period of time, and is hardly clicked on by users. This shows us the difference between games that are hardly clicked on, and games that are just newly added to our dataset.

One of our failing points is that we were a little confused on how to incorporate these scores into the results we return. In class, we don't believe we talked about how to use this feedback to return better results, or if we did it, might have been briefly. So we can collect data on games, but the feedback is not being incorporated into our recommendation algorithm. If we could possibly get some ideas in the feedback that would help us greatly even though we won't be able to implement it at this time. We didn't want to incorrectly implement an algorithm to use this feedback in our recommendation system and have it affect the results we are returning in a negative way.

```
{
  "227200": 0,
  "8350": 1,
  "255370": -2,
  "94303": -1,
  "67000": -1,
  "35700": -3,
  "200960": -1,
  "206060": -1,
  "207570": -1,
  "31220": -1,
  "218740": -1,
  "25900": -1,
  "47540": -1,
  "245430": -1,
  "239070": -1,
  "37400": -1,
  "113200": -1,
  "210230": 1,
  "219200": -2,
  "358310": -3,
  "80360": -1,
  "37260": -1,
  "212050": -1,
  "231160": -1,
  "358320": -1,
  "20820": -3,
  "29120": -1,
  "38740": -1,
  "358190": -1,
  "405640": -1,
  "332200": -1,
```


Figure 6 – Example of game click feedback

Game Correlation Analysis

When looking at the data for game correlations we found something very interesting in our data. There are some games that are free to play, and some of games that come in a Steam bundle for about $50. That $50 gets you about 15 games. So we found that almost all of our users own these free to play games, and a high majority own these bundle games. Therefore our algorithm counts all these users owning these games, and ends up correlating them. We thought that if we were to recount the correlation of games, we would have to ignore the free to play games from our correlations all together, as well as the bundled games. This would clear the clutter out from our results from the correlation process, allowing us to more clearly see which games are highly correlated.

We also find that some of the free to play games are correlated with some games that aren't free to play. This may be a good indicator for game correlation. If some users have played some of the games that are free to play, they might also like a game that is not free to play. We decide to not match a free to play game with another free to play game to try and remove some of the useless correlations that may occur. This allows us to match games that are not only free to play, and are highly correlated between our users. This is important because games that are not free to play are not owned by everyone. This allows us to deduce that games owned by a large majority of people that are payed games may be highly correlated due to the money investment needed by the user. When a large majority of users own the same pair of payed games it should be assumed that if a person plays one of those games they may like the other game as well and vice versus.

```
"deathmatch classic-ricochet": 3363,
"deathmatch classic-halflife": 3364,
"deathmatch classic-halflife blue shift": 3364,
"halflife opposing force-ricochet": 3363,
"halflife-halflife opposing force": 3364,
"halflife blue shift-halflife opposing force": 3364,
"halflife-ricochet": 3363,
"halflife blue shift-ricochet": 3363,
"halflife-halflife blue shift": 3364,
"counterstrike-halflife 2": 1657,
"counterstrike-counterstrike source": 1861,
"counterstrike-halflife 2 deathmatch": 1768,
"counterstrike-halflife 2 lost coast": 1782,
"halflife 2-team fortress classic": 1656,
"counterstrike source-team fortress classic": 1860,
"halflife 2 deathmatch-team fortress classic": 1767,
"halflife 2 lost coast-team fortress classic": 1781,
"day of defeat-halflife 2": 1657,
"counterstrike source-day of defeat": 1861,
"day of defeat-halflife 2 deathmatch": 1768,
"day of defeat-halflife 2 lost coast": 1782,
"deathmatch classic-halflife 2": 1657,
"counterstrike source-deathmatch classic": 1861,
"deathmatch classic-halflife 2 deathmatch": 1768,
"deathmatch classic-halflife 2 lost coast": 1782,
"halflife 2-halflife opposing force": 1657,
"counterstrike source-halflife opposing force": 1861,
"halflife 2 deathmatch-halflife opposing force": 1768,
"halflife 2 lost coast-halflife opposing force": 1782,
"halflife 2-ricochet": 1656,
"counterstrike source-ricochet": 1860,
"halflife 2 deathmatch-ricochet": 1767,
"halflife 2 lost coast-ricochet": 1781,
"halflife-halflife 2": 1657,
"counterstrike source-halflife": 1861,
"halflife-halflife 2 deathmatch": 1768,
"halflife-halflife 2 lost coast": 1782,
"halflife 2-halflife blue shift": 1657,
"counterstrike source-halflife blue shift": 1861,
"halflife 2 deathmatch-halflife blue shift": 1768,
"halflife 2 lost coast-halflife blue shift": 1782,
"counterstrike source-halflife 2": 1590,
"halflife 2-halflife 2 deathmatch": 1624,
"halflife 2-halflife 2 lost coast": 1658,
"counterstrike source-halflife 2 deathmatch": 1682,
"counterstrike source-halflife 2 lost coast": 1682,
"halflife 2 deathmatch-halflife 2 lost coast": 1750,
"counterstrike-counterstrike global offensive": 1186,
"counterstrike global offensive-team fortress classic": 1186,
"counterstrike global offensive-day of defeat": 1186,
"counterstrike global offensive-deathmatch classic": 1186,
"counterstrike global offensive-halflife opposing force": 1186,
"counterstrike global offensive-ricochet": 1186,
"counterstrike global offensive-halflife": 1186,
```


Figure 7 – Example of correlations and scores

Query Results

When returning a results list to the user, we had some interesting findings. Since we pre-score all of the games before a query happens the results returned to the user can become stagnant. For example if you search an action game, and we only return you the top 10 action games, and you search another action game and we return only the top 10 action games the results list will be the same. Therefore to throw in some variance of results returned we return 5 results that are rated 90% and above, 5 results that are between 80% and 90, and 5 results between 70% and 80%. We also throw in some variance on deciding which 5 games are returned from each scoring block. Doing it this way allows us to recommend good games from each scoring block, while still allowing us to return games that are scored very highly.

We have done some user testing with some friends of ours, and the results were quite interesting. Our friend Shane searched for "doom ii", while looking through the results list we returned him he found a game he liked called "broforce" which was rated 0.72. This shows that just because it was rated 0.72 it just isn't irrelevant to the user, since 0.7 is still a high score. Emil searched for "Dota 2", he also found a game he liked called "waking mars" which was rated 0.97 one of the highest rated games in the results that were returned. Trent searched for the game "off world trading company", he selected a game called "Rome total war alexander" which was rated 0.78. Even though our testing group was small, each one of them was able to select a game that they thought might be interesting to try based on the result set that we returned to them.

In addition to returning results that score above 70%, we decided to return one extra game if the game that was searched for is highly correlated with another game. For example if you search "CounterStrike" a game highly correlated with that is "Counterstrike global offensive". These two pairs of games are owned together by 1186 users out of 3400 of our user data set. So since more than a third of users own CounterStrike with CounterStrike global offensive we would return CounterStrike Global offensive as well with our recommendation results.

Another feature we implemented into our recommendation system is that we can fine-tune the results we display to each user. We accomplish this by having them put in their own Steam ID, which allows us to query the Steam API and retrieve the list of games that user owns. Since we now have some good data about our user, when returning results to the user we can make sure not to return games they already own. This is an important feature to our recommendation system, because it allows us to return more accurate results. If we return games that the user already owns then those games are technically not meeting the information need.

Related works

Comparison of recommender systems

One problem with recommendation systems is that nobody has compared them. There are numerous algorithms to help you write a recommendation search, but nobody yet has put them side by side and gave in depth analysis of each algorithm. Recommendation systems are all about personalizing the search experience for the user. Most recommendation systems do this by pulling information from profiles and past searches to find useful trends. The paper presents one of the problems with comparing recommendation systems, "A consensus still has not been reached on the characteristics that should be evaluated. The most common tendency is to evaluate the accuracy or precision of the algorithm. Herlocker et al. [2004] identify three types of metrics to measure the quality of an algorithm." So the three characteristics they chose to evaluate recommendation systems is Prediction accuracy, Classification accuracy, and Rank accuracy. I think when evaluating our own recommendation system, reviewing these characteristics would be a good idea to possibly improve our recommendation system. A problem with comparing evaluations from several works however, is the way the dataset is split up into evaluation, and a training set. Some researchers may split the data up differently, and when comparing the evaluations would drastically throw off results when comparing the recommendation systems.

The paper also talks about quantity of results returned which I found interesting. They say "Coverage, corresponds to the percentage of items the system is able to recommend. Coverage can be used to detect algorithms that, although they present good accuracy, recommend only a small number of items. These are usually very popular items with which the user is already familiar without the help of the system". This means that when returning results you want a high number of results returned all with good accuracy. This kind of system would be most desirable since you are returning more results that the user may not have known about, and is relevant to their informational need. Overall I thought this paper was very useful, and I hope to use some of its insights when designing our recommendation system for video games.

A Market Based Approach to Recommender System

Wei, Moreau, and Jennings have based their recommendation system architecture upon a free market strategy. One variation of the system is based off of an auction system, they break their recommendation system into a seller, an auctioneer, and recommending agents. Initially the system feels as if it is backwards; the seller is the person using the system and the recommending agents are the bidders in the auction. However, this is not the case. The auctioneer or the recommendation system picks recommendations from the agents and shows them to the seller. The seller picks from the results and the auctioneer rewards those agents who made the recommendation. The whole premise of the system is that by rewarding the agents, the agents that are chosen more often are continually rewarded and their recommendations from those agents are displayed first and are seen to be more relevant to the seller's query. The authors found that there is no universal auction design that is applicable to every query, and so they could not use only this design.

They needed a "market" with some kind of standard properties. They included Pareto efficiency, Social welfare maximization, individually rationality, convergence, an Effective shortlist in decreasing order of user perceived quality, Clear incentives, Stability, and Fairness. The Pareto efficiency states that it is impossible to make one solution better than another unless another such solution is worse off. In this case, the solution is represented by the recommendation agents. An agent cannot be better than another agent without making another agent worse off. Social welfare maximization is a way to rank the recommendation agents in terms of usefulness. Individually rationality is a measure of worth. The recommendation agents weigh whether or not it would be profitable to participate in a recommendation. Since in this system the agents can be punished for not having a good recommendation. Convergence helps the agents see whether or not they have a profitable margin for recommendations. After many auctions or queries, the system may tend to converge. If it does, this gives the agents a margin to give relevant recommendations. The effective shortlist in decreasing order of user perceived quality is the overall goal of the free market. If these lists can be generated fast and efficiently, the search engine is a good one. Clear incentives are for the agents. It gives the agents a reason to give recommendations. Stability and Fairness are good market protocols, and are used to monitor the behavior of the overall system.

The concept of a free market recommendation system is very interesting. It makes sense that the most popular "bidders" become more easily recommended. However, this has an issue where it could start to ignore some bidders regardless of how relevant the recommendation is to the query. This might be solved by the convergence margin, but if it does solve it, this is not blatantly obvious.

Reliable Medical Recommendation Systems with Patient Privacy

Hoens, Blanton, Steele and Chawla wanted to create a system that recommends physicians to patients based on the health conditions of the particular patient. The system maintains a list of physicians and health conditions. Each physician is rated for the satisfaction of treatment by past patients for each of the health conditions. A patient who is interested in obtaining a physician recommendation will be able to see a list of physicians who are best ranked for that ailment. Should the patient be interested in seeing a physician for a combination of health conditions, the system will show results for the best physician for all of the ailments.

To normalize ratings and results for the physician, the authors used an average of all ratings for each specific physician. All ratings to a physician cannot be traced back to the patient that rated him or her. The system had to be reliable, as such, the authors implemented a way to deal with dishonest and malicious ratings. The il-ratings that are detected are either prevented or compensated for.

This type of recommendation system is very generic in that it recommends based on prior patient input. Each physician is scored and these scores are totaled and computed to give the best recommendation to the patient. What makes this project stand out is the patient privacy and the detection for dishonest or malicious ratings.

Netflix Recommender System

Netflix has been a huge source for improving recommendation searches. Netflix claims that 80% of its hours watched are choices from its recommendation system, and the other 20% is from its search. How they accomplish this is by tailoring each search to its user. This makes sense since Netflix has a lot of data of shows you've watched, series you've completed, shows you've stopped watching 30 minutes in. By tailoring it to the individual user it allows for a more accurate retrieval of data that fits the users needs.

How this applies to our projects is that one of the ideas we thought would be good is retrieving the users steam games. What we can do with this is ensure that we don't give a game suggestion that they already own. Since the user already owns the game it would defeat the purpose of our whole project of suggesting new games the user isn't aware of.

When using Netflix I noticed an interesting recommendation section called trending now that I thought was cool. When reading the paper I kept it in the back of my mind wanting to know more information about how this section worked. Questions such as: is it global trending? Or local trending? This would make a massive difference, because people in Spain may prefer different TV shows than people in California. From what I could deduce from the article they do local trending, although there are still some struggles doing trending this way. For example when Netflix is now available in a new country there is no data to support what shows it should recommend for this section, or any section to be fair. They have no data on what the country likes to watch. So if there is one really weird person who watches a wide range of bizarre unrelated content that could drastically throw off the recommendation algorithm developed by Netflix. Although over time this problem will sort itself out, in the short term recommendations may not work as intended giving users TV shows they may not want. I do feel like a model that gives many types of recommendations is a good one. Netflix itself has boasted that 80% of the hours watched are from shows that Netflix has recommended. So maybe having multiple recommendations for each user for video games could work too. We could have a recommendation for action games, adventure games, and puzzle games. This

would be a step towards a recommendation model that doesn't use queries, but instead recommends games based on type of game and games the user has played in the past.

Reading how Netflix took interest in what shows people watched in full or partially to better give recommendations of TV shows gave me some ideas that could possibly be explored if given enough time. Since we can get games that a user owns on steam we could theoretically do what Netflix does. We could use the data to say that since this person bought game1 and game2 those games might be correlated to some effect. The difference might be is that people don't need to pay additional money to watch more TV shows, while each game costs money. So money might be a factor to account for as well. Since maybe one person will only buy a game if it is $20 or less, while maybe someone else doesn't have any monetary restrictions.

Overall I really like Netflix's model on recommendation systems, and given enough time I'd love to implement some of the features they have in their application. Although I feel the time constraint we face we won't be able to do so for all of it.

Recommendation systems with complex constraints

This paper talks about comparing complex objects with constraints. Within their context they are doing research on selecting courses at Stanford University. They want to select courses that meet certain perquisites, and that are also desirable to similar students. This seems like a simple task, and for humans it is, but for computers this is where complexity sets in. In order to check all these requirements increases the time complexity of the algorithm. Which henceforth makes it difficult to give recommendations when your algorithm to recommend classes takes a long time to processes the information.

How this relates to our project is we are somewhat doing a small subset problem of this one, where a user may require the game meet certain criteria such as: multiplayer, first person etc. So we must only suggest games that meet these criteria or we would be giving the user information back that is irrelevant to the query need.

I found the way they talk about scoring classes to recommend interesting as well. Courses that have some overlap they decide to penalize, and courses that complement each other they decide to reward. I think combining this tactic of scoring with recommendations that lay out a plan over quarters and semesters goes very well together. Being able to determine classes that go well together and classes that don't, and then be able to lay out a class plan over the time you go to a school would be very key in helping students graduate on time. I think some downsides to this is that for computer science majors it should recommend the same courses to every computer science major for the year of school they are in. Since all freshmen computer science majors will more or less have the same prerequisites, and therefore impacting a series of classes causing an overflow of desire for some classes. I mean when I was slugging through the college curriculum a lot of the choices I made for classes had to do with what class still had room in it.

The people who wrote this paper believe that complex restraints come in other forms for recommendation scenarios. One of their examples is "Say we need to recommend 20 movies for screening at a movie festival. In addition, at least three movies of each genre that is,

horror, comedy, romance, action, need to be screened. In this case, there is no upper bound on the number of movies of each genre that can be shown. This situation directly corresponds to a range version of the core model of requirements; that is, requirements of the form take k1-to-k2 items from S with movie selection for each genre representing one sub-requirement (lower bound on number of items per sub-requirement is three, upper bound ∞). The global requirement of 20 movies can also be captured in our model". This is a prime example of a complex recommendation scenario with multiple constraints. I believe some of their models could be adapted to our project to help with recommending games based on genre, company who made the game, price, etc.

A Multimedia Recommender System

Albanese, D'Acierno, Moscato, Persia and Picariello proposed a system that recommends art based on a query. Their recommendation strategy was to use the following tools: an Item Manager, Session Logger, and various computational modules. The Items Manager is basically a large index of art pieces. Each art piece is an image that contains raw data, a short description, and metadata. The Session Logger is a module that stores information about all of the items viewed in the current session. This allows for the system to check its recommendations against the items that the user seems interested in.

The Recommendation Engine uses a bunch of different computational modules to produce a set of recommended objects that are ordered in decreasing utility. The Browsing Matrices Computation Module takes into account both the browsing data from all users as well as the browsing data from a single user via the Session Logger. This module allows the recommendation engine to cross reference the data gathered to the list of potential objects and make a judgement to which objects are more relevant to the query. The similarity Matrix Computational Module computes the similarity of objects. If objects are too similar, the Recommendation Engine might not want to display all of those objects. The Candidate Set Selection module computes a subset of objects from the potentials that are more likely to meet the requirements of the user based on the query. The last module, the Rank Computation Module picks all of the candidate objects for recommendation.

Conclusions

Some conclusions that we think can be drawn are, there is more than one way to make a recommendation system. We decided not to use TF.IDF and make our queries simple game names. You can't use TF.IDF when just using the game name as the query due to the fact that other reviews probably don't mention the other games often if at all. In the future we could also display games how Netflix displays movies, which is the no query format of recommendation system. We could do this by simply having the genres for action, adventure, strategy etc… and displaying results that way. By doing this we would fully eliminate the query.

When trying to determine how to score the games we struggled to find a good solution. We knew the key was to do some sort of pre-query scoring, but given that we only have star ratings and reviews it seemed a little difficult. We were really centered on the ideas we learned in class such as TF.IDF, and had to be creative when trying to devise a new strategy that would score the games accurately. Even though our first approach of having two lists of words positive, and negative failed. We made the realization that when people write reviews they like to use positive words negatively for the most part. It might be because when writing something negative, if they use positive words it sounds as if they are saying something nicer.

We believe our best feature is when we decided to create an algorithm that correlates two games together to better recommend one of them. Matt got the idea from the Netflix related works section, where if lots of people watch the same movie they eventually become correlated. So to adopt that idea to our recommendation engine we had to get a large list of users from steam and write an algorithm that pairs all the games together that have the same genre. This algorithm is expensive, but since it only has to be run once pre-query, we thought that it is an okay sacrifice. You can see part of that list at Figure 7.

Refining our recommendation system was a little tricky, we had another idea though. Since we can get a list of user's games from steam through an API call, we decided to allow users to input their own Steam ID. What this allows us to do is query their list of games from the Steam API. Since we now have a list of games that the user owns, we can simply eliminate games he/she already owns from the list of results we are going to return. This allows us to not

recommend games to the user that they may already have, which would be a bad recommendation if returned to the user.

After completing this project I feel we have a good set of features explained in this paper that make our recommendation system great. We do have some faults that may need to be fine-tuned in the future for better results, such as incorporating the user feedback results into our recommendation system. However, we believe we can defend the decisions we made in this project to our fullest capabilities, and are really proud of what we were able to accomplish as undergraduates in a graduate level Search Engines class.

References

Zheng Wei, Yan, Luc Moreau, and Nicholas R. Jennings. "A Market-Based Approach to Recommender Systems." *ACM Transactions on Information Systems* 23.3 (2005): 227-66. Web.

Hoens, T. Ryan, Marina Blanton, Aaron Steele, and Nitesh V. Chawla. "Reliable Medical Recommendation Systems with Patient Privacy." *ACM Transactions on Information Systems and Technology* 4.4 (2013): 67:1-7:31. Web.

Albanese, Massimiliano, Antonio D'Acierno, Vincenzo Moscato, Fabio Persia, and Antonio Picariello. "A Multimedia Recommender System." *ACM Transactions on Internet Technology* 13.1 (2013): 3:1-3:32. Web.

Cacheda, Fidel, Victor Carneiro, and Diego Fernandez. *"Comparison of Collaborative Filtering Algorithms: Limitations of Current Techniques and Proposals for Scalable, High-Performance Recommender Systems."* (2010): n. pag. Web.

CARLOS A. GOMEZ-URIBE and NEIL HUNT, Netflix, Inc. *"The Netflix Recommender System: Algorithms, Business Value, and Innovation."* (2015): n. page. Web.

ADITYA PARAMESWARAN, PETROS VENETIS, and HECTOR GARCIA-MOLINA. *"Recommendation Systems with Complex Constraints: A Course Recommendation Perspective"* November (2011): n. page. Web.

Minqing Hu and Bing Liu. "Mining and Summarizing Customer Reviews." Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-2004), Aug 22-25, 2004, Seattle, Washington, USA,

Bing Liu, Minqing Hu and Junsheng Cheng. "Opinion Observer: Analyzing and Comparing Opinions on the Web." Proceedings of the 14th International World Wide Web conference (WWW-2005), May 10-14, 2005, Chiba, Japan.