**Title: Comparacter**

**Date: 14 May 2018**

**Intro:**

Have you ever been watching a movie or a TV show and found one or two specific characters particularly appealing? I’m pretty sure anybody that watches TV can identify with that. So, a group of three of my friends and myself built a system which, given a character from a specific movie, will return a list of other similar characters in different movies! (N.b. This was a class project, done as part of CS 4300).

Try it out here.

**Getting Started:**

The first steps in the project as a whole was to collect our data and process it in a way that was helpful to us. This resulted in us collecting a mammoth amount of data, including about 1300 different movie scripts, 25000+ movie reviews, and 5000+ lines of character and movie metadata. Of course, this would be a lot for one person, so at this point we planned out exactly what we needed to do to complete the project and divided up the work. I was in charge of movie script processing, the main part of the character similarity/comparison metrics, and the Rocchio relevance feedback update. Other portions of the project included front-end design, back-end maintenance, and the rest of the data processing.

**Data Processing:**

This was undeniably the lengthiest portion of the project. We had 3 different datasets with movie scripts in different formats, so the first job was to process them so that they were in a usable format going forward. I chose to do this by creating lists of dictionaries (HashMaps) for each movie, where each dictionary consisted of the line itself and the speaker. This allowed for easier comparison between characters and movies in the future. After this, I used Sk-learn to do some TF-IDF vectorization (7500 features) in order to start doing character similarity.

**Similarity metrics, NLP, and Rocchio Relevance Feedback:**

After processing the data, I had to figure out how to accurately compare characters to get the most similar based *entirely* on their words – the scripts had no indication of expression or tone. For the first demo, we just had to have a working similarity metric, so I chose to use a basic cosine similarity. However, while the results were acceptable with this metric, they weren’t stellar. As a result, we tried to incorporate more complex NLP techniques (notably SVD) which would yield better and more consistent results.

We ended up using a Stanford created NLP library called Empath to ease the process. Empath helped perform SVD and produced standardized topic/sentiment tags when given a line. What I then did was to produce a count of empath tags per character, normalized against the total tags per movie, which avoided longer movies having more tags and thus being more ‘similar’ to others. I felt that cosine similarity between these tags wouldn’t be particularly helpful, as there was a small and finite number of tags (~200). Therefore, I chose to use general Jaccard similarity to compare the traits for each character to one another.

My final task was to implement Rocchio Relevance feedback. Rocchio is essentially a way of refining search queries based on some sort of user feedback about the given results. The way that we chose to do this was to display the top 10 character traits and their importance to that character (in the form of a radar chart), and allow the user to toggle certain tags’ weights in the query.

**Challenges:**

***Big data, Algorithmic complexity: Time and Space***

There were two separate sections of algorithms I had to worry about – the first, data pre-processing, comprised of one time calculations dealing with larger, non-runtime problems. The second, live query response, deals with runtime functions.

This was my first experience dealing with such a large dataset (the dataset, purely comprised of text documents, was ~1GB, and the original Cosine similarity matrix was > 500MB, as it had about 128,000,000 entries). As a result, I thought that, even for pre-processing, just running the first iteration of code I could think of would suffice, even if it took a little longer. Unfortunately, I ran into a road block when trying to calculate the Cosine similarity matrix – by my calculations, it would have taken about 20 straight hours of full CPU usage on my laptop.

Obviously, this wasn’t a feasible process, so I had to think about another solution. Re-reading the fundamentals of what a Cos-sim matrix is, I realized that this process could be solved by simple linear algebra. By multiplying the TF-IDF matrix with the transpose of itself, then dividing by the norm matrix, I achieved the same result as calculating each individual score, but having saved 19 hours and 59 minutes.

I later ran into a similar issue on the query side of things. The built-in python sort (returning highest scoring characters) resulted in around a 10 second query lookup time, which obviously was not acceptable. So I decided to, after experimentation, use quicksort. I used this over variants like merge sort because it is objectively the quickest, and being unstable is not relevant here as movies were processed in an arbitrary order in any case.

**Fire and Shadow**

**Intro:**

A few months ago, a friend of mine (Ryan Slama) and I sat down and decided to make a game, just for a fun little programming project. We spent a while ‘spitballing’, and, after shooting down a number of ideas, came up with our idea, that would be both challenging to create and provide good gameplay and replay value. The concept was, in essence, a tower defence/survival/RPG hybrid, top-down game. It proved to be a fun and interesting journey!

Check it out here.

**Getting Started:**

Neither of us had ever produced a game before, so our first challenge was which language to do it in. We chose to use Python, but Java, Javascript, and C were also proposed. In hindsight, it would have been better to do in Java or something more strictly geared towards OOP, because the code got unwieldy quickly. That night we went from about 10pm-4am, and got a decent amount accomplished. I focused more on the game mechanics and started reading up on PyGame and general good practices in game programming, and Ryan, being eminently better than me at design, did most of the character and graphic design, although I did do a few elements here and there, especially things related to the HUD.

We planned to implement multiple character types/ classes, but decided to start with a Wizard for now. By the end of our session, we had basic character movement, HUD, terrain, and concept fairly set out. It was looking good!

**Progression of the game, and where it stands now**

We never really sat down together after that for extended periods to work on it, but instead had a shared Git todo list, from which we’d both take tasks and separately implement them. My main ones included implementing major character mechanics, interaction with the board, collision/placement detection, some HUD elements, and tweaks to the enemy AI. Slowly, features like the basic Wizard attacks, placement of traps, towers, and other supplements to the player became implemented. We then started introducing gold as a separate metric to score, added basic enemies (which Ryan lovingly modelled after my LinkedIn picture), and a much needed development/debugging mode, which essentially rendered total map vision, invincibility, max gold, and a few other features that made debugging far easier.

Slowly, more and more small features began getting added, and our todo list was constantly updating with completed tasks and new ideas. We had introduced different enemy types, a wave system with scaling difficulty, and more advanced pathfinding techniques. We had a relatively solid game!

It was at this point that prelims, classwork, and other commitments and projects started to catch up to us. The game, while still technically in development (but of course, fun & fully playable), has had its development slow down considerably in recent months. It’s been an incredibly enjoyable project, opening up an entirely new section of computer science and introducing me to new libraries and skillsets.

**Challenges**

**Code structure and hierarchy**

Overall, there was nothing really challenging of note about making the game itself – figuring out ways around problems were fairly simple and Ryan was great to work with, so we were always making solid progress. The challenging part, however, was organizing and documenting the code. Python, as awesome of a language as it is, does not naturally lend itself to organization and structure like Java, due to its mix of OOP and functional features. As a result, when the game grew rapidly we got caught with ‘spaghetti code’ everywhere, with functions being strewn everywhere, over-reliance on global variables, and difficulties connecting parts of the code that we wrote independently (due to documentation issues).

The first step is always realizing there’s a problem – so when we finally accepted it, we began by going through and changing functions to have clear and specific definitions, and reducing dependence on global variables by instead passing them as arguments. Finally, we split the code from our ~4 separate modules into 13.