Dylan Hayton-Ruffner
Computational Creativity
Prof. Sarah Harmon

SCRIPTY

Overview: SCRIPTY applies deep learning to the task of script structure generation i.e. the sequencing of scene changes, stage direction, and dialogue sections of a film script. The name scripty was chose because program is writen in python and the output of the program is "script-like" (a framework). The system utilizes a customized scrapy CrawlSpider to extract all the scripts from the tv show *Seinfeld*. Utilizing rules embedded in the returned html documents, the program converts each raw script into a list of 3 essential elements (stage directions, setting changes, and dialogue lines). The resulting lists are convoluted with a window size of 100 resulting a set of 50,000+ vectors. This set is divided into input (v[:99]) and target (v[-1]). A neural network is then trained on these vectors to predict a script element based on preceding elements. The network is then utilized to generate the sequence of elements that make up a *Seinfeld* script.

Setup/The SCRIPTY interface: To start a SCRIPTY session run scripty.py with python3+ located in the directory "script". This will load the scripty main interface. From the prompt the user can generate sequences, realize (produce full scripts with text) them with a mere generation method, write these files to text, display evaluations of each script, retrain the net, and load models. The help command displays usage info. Generated works are written into the "works" directory.

System Design: Scripty focuses heavily on script architecture, i.e. the set and sequence of scenes (dialogue, directions, and setting changes) that a narrative is realized in. Surprisingly, I could not find a lot of research on this subject. There is a lot of work on how to build an engaging narrative structure for a movie, play, tv show[1][2]. These systems develop high level plot architectures - graphs, trees, lists of story elements. However, once a plot structure is developed, these systems still need a sequence of elements to realize their screen play in. In *Xu et. al.* a deep learning neural net is used to convert from narrative elements to specific dialogue elements[3]. This system hopes to take a similar approach and bring it into the realm of tv sitcoms. Automated script generation is the holy grail of tv and genre movie production. These forms of media have similar plots that map over the same

[1] Story Generation with Crowdsourced Plot Graphs, Li et. al.
[2] Evolving Stories: Tree Adjoining Grammar Guided Genetic Programming for Complex Plot Generation, Wang et. al.
[3] Towards Generating Stylistic Dialogues for Narratives Using Data-Driven Approaches, Xu et. al.

scene structure in a script[4]. Thus, if a system can learn the pattern of a genre script, it can reproduce a structure that will likely work for plots of that genre. Thus, a huge resource to screen writers would be a framework generator, a way to compile a set of script elements in which to realize their plot. This system hopes to take the idea of neural network sequence generation in *Xu et. al.*[5] and use it learn the basic formulaic sequence of script elements in a *Seinfeld* screenplay. With a deep understanding of this sequence, the system will be able to create novel script structures that fit within the *Seinfeld* medium and effectively realize its plots. The script is broken into 3 elements: Dialogue, Setting, and Stage Direction. This simplicity allows the system to truly learn the rhythm and pattern of a script. This system is forced to learn scene boundaries, the rhythm of dialogue in scenes and the correct placement of non-verbal actions. The generality of these elements also give a plot generation system greater leeway when working with the produced sequences. The system has 4 components: data scraping, data preprocessing, training, and generation.

Data Scraping: In order to collect enough data to train a neural net, this system utilizes a scrapy spider to extract data from www.imsdb.com. The system extends a scrapy CrawlSpider and uses it to find and download the html from all pages of the show Seinfeld. The spider uses a set of rules to crawl the page. When it follows a link that has a script, it downloads the html into a local repository.

Data Preprocessing: Data scraping yielded 176 raw html files to be processed. The first step is this process is done by preprocess_html_scripts.py. This file extracts the script section of the html and process it, marking each section as belonging to a specific script element: Stage Direction, Setting, Dialogue. The format of the script within the html was very convoluted. Much of the actual text is contained in the tail of b tags. A rough set of rules could be followed to determine the type of each text: If the tag has text, it is a character name. If that tag has text and it has a tail, then that tail is dialogue associated with that character. If the tag has no text it is a setting or stage direction. If the tail is encased by "()" then it is a setting otherwise it is a stage direction. Unfortunately, the last rule sometimes labels dialogue as stage directions leading to a somewhat inflated amount of stage directions detected in dialogue sequences in the scripts. As the file is parsed, a new html tree is built containing only labeled script elements in order. The new tree is written to the directory "processed_html_scrips". The next part of the process is done by "parse_html_scripts.py". This file takes each preprocessed html file and parses it into

[4] Cracking the Sitcom Code, Noah Charney, https://www.theatlantic.com/entertainment/archive/2014/12/cracking-the-sitcom-code/384068/

[5] Xu et. al.

a Script object. The main field of the script object is a list called “script” which is a sequence of all the elements in the script. The different elements are represented by the classes Character, Setting, and StageDir. The Script class also supports a get_training_data function which passes a 100 unit window over the data, extracting vectors. In parse_html_scripts.__main__, each of these vectors is written to a csv file. This process is repeated for all 179 scripts. 50,000+ training examples are extracted.

Training: Training is conducted by scripty.py. The file has a two main classes CSVDataProcessor, which handles parsing and preparation of data for training, and ScriptSeqNet, which uses a tensorflow.keras.Sequential model to generate scripts. CSVDataProcessor takes the file written during preprocessing and loads it into a python List. It then divides that list into training and test sets. These sets are passedto the ScriptSeqNet. It loads the data, builds the model, and trains on the data. The model utilizes a 99 node input layer, 2 dense, hidden layers of 30 nodes with sigmoid activation functions, and a 3 node softmax, output layer. The system trains for 200 epochs with gradient updates every 10 samples, and reaches +90% accuracy on the training set and +80% accuracy on the test set.

Generation: Generation is handled by scripty.py with the ScriptSeqNet class. The generate method loads a model and uses it to generate a script. It picks a random sequence from its training data to use for “inspiration” and then begins to generate, predicting elements based on the last 99 elements. After reaching a preset length (400-avg Seinfeld script length) the script is returned with the inspiration set removed. This process is mediated by the Generator class. It takes a ScriptSeqNet and a LineGenerator class and uses them to develop a sequence of elements for the script and then realize that sequence with real dialogues, settings, and stage directions. The LineGenerator is the weakest part of this project. Because the rest of the system is robust, I only had time to develop a mere generator for this element. The LineGenerator iterates through the script and fills in each element with a randomly drawn matching element from its knowledge base of *Seinfeld* script lines. Thus, the scripts are not very valuable for their plot or textual content. However, the focus of this project was to develop a system to help robust plot and text generators realize their creations. Thus, combined with one of these other programs, this sequencer would be very effective. The LineGenerator writes these realized scripts to a file. The Generator object also contains several evaluations methods. It creates a originality score by checking the similarity of the script to other scripts in its corpus. It also checks domain competency by looking at differences between its works and other’s works. Both of these are implemented by looking at each element in the two scripts (new and old) and comparing them. If they are different, a value of one is added to the score. Thus, high scores indicate lots of difference. High originality scores are better. Low domain competency scores are preferable. A more robust scoring system that looks for patterns within the scripts would be much more effective.

Critical Limitation: Although I have already discussed limitations in the section above, there is one that is very important. Although there is some structure to the scripts parsed from www.imsdb.com there is enough ambiguity between stage direction and dialogue that a significant amount of dialogue segments were mislabeled as stage directions. This can be seen in the textual scripts output into the works directory. Many of the works contain stage directions that are clearly dialogue segments. This skewed training of the sequences as well, leading to far too many stage directions in the produced works. I spent a lot of time trying to solve this ambiguity but didn't have enough to time to implement the methods I devised. The best way to solve this issue is to create a system that can classify a chunk of text as setting, dialogue, or stage direction. Setting is trivial as all settings are framed in "()". Thus, differentiation between dialogue and setting is the critical problem. I was unable to solve this problem.

Computational Creativity: While there are many definitions of creativity this analysis will focus on three key metrics proposed made by Jordanous in her SPECS criteria[6]:

1.) **Dealing with Uncertainty**
    a.) This system is very good at dealing with uncertainty in input. For any set of inputs the neural net is capable of producing an output. This allows a script structure to be generated from any inspiring data set. Moreover, when encountering new sequences the system can utilize knowledge it learned while training to generate elements despite being in a new situation. This ability to deal with ambiguity makes the system more robust and creative.

2.) **Thinking and Evaluation**
    a.) This system utilizes a robust evaluation system but lacks a thinking element. The evaluation system is two fold. Each work generated is compared to other works generated by the system in the past. This is the work's creativity score, the average distance between the work and all others produced by the system. This score is slightly flawed

[6] Information from lecture and class page

however. Different scripts have different inspirations so they should be different. However, this project operates under the assumption that Seinfeld scripts converge in output and thus this problem should be minimal. The next element of evaluation is an assessment of domain competency. In this test, the script is compared to real Seinfeld scripts. The number of differences are tallied and averaged. This also suffers from the limitations of the other metric. The comparisons made are within the first 200 elements of a real script. The scripts generated are not necessarily generated to mimic the start of a script. Thus, both of these metrics would benefit from a more robust comparison or distance computation, one that detected pattern similarity rather than euclidean distance. However, they do give the system a good idea of how valuable its output is. The purpose of the system is create novel scripts within a predefined genre. These metrics measure this goal well.

b.) This system evaluates but does not use its evaluations in real time. It does not think critically as it creates. It makes and then evaluates. This is due to the nature of neural networks. It is very difficult to understand how one is operating and thus difficult to affect its output mid process.

3.) Domain Competency and General Intellect

a.) This system clearly values domain competency. By picking one specific type of script from a specific show, the system learns how to work within the set of rules that describe this genre. Moreover, the system has a method of understanding a domain embedded in its domain competency metric. It can judge and make evaluations based on an understanding of all the works of the genre. Thus, the system has generalized knowledge of the genres rules (neural net) and a wide knowledge based associated with the genre.

b.) I chose to combine these two SPEC themes in my analysis because the emphasis on domain competency affects the system's general intellect. In one sense the system can be said to have low general intellect. It focuses on one domain, and emphasizes a very specific part of that domain. However, the use of the neural network hints at a greater understanding hidden in the system. While it is hard to determine, the system on some deeper level has learned to understand the Seinfeld script sequence. It can take a variety of inputs and produce interesting sequences as a result. Thus, it does a have a great amount of generality *within* the medium chosen. It can create works with a novelty factor of ~200 differences per script on average and a closeness of ~60 differences to most scripts of the genre.

Personal Challenges: I challenged myself on my assignment by choosing a project in which I had to learn several complex libraries and new techniques:

Scrapy: Scrapy is a python library for scraping webpages. It provides a variety of methods to utilize but the most applicable API to my problem was the CrawlSpider. I had to learn how to set up the scrapy environment and extend the CrawlSpider class to build a spider that could scrape the data I was looking for. This involved reading scrapy documentation and understanding the pipeline of data in the scraping process.

CSV: CSV is the python library for reading and writing csv files. This was the easiest of the libraries to learn. It has a relatively simple API and my applications for it were not particularly complex.

ElementTree: I used element tree to parse and build html simultaneously. This was difficult because I had to understand how the library represented a tree and how to add and build html using that tree.

Beautiful Soup: I used bs4 to quickly parse and prettify() html. It was very useful and relatively easy to learn.

Tensorflow.Keras: I used the tensorflow.keras.models.Sequential object to implement my neural network. This was the most difficult library to learn. I had to understand how a model is built, how to format data for input into the model, and what optimizers and activation functions to assign to each layer. Building the network was a new challenge. There is no perfect architecture for a neural net so I had to experiment a bunch. This ambiguity was very challenging, especially since the API was so new to me.

I also had to work with and structure code in new ways:

Classes: This system has a ton of moving parts and to correctly package them all I had to develop and work within python’s OOP more than I have had to in the past. The majority of this involved learning to extend complex classes in libraries and thinking about how to assign roles to a complicated set of classes.

Information Pipelines: This project requires that I pass large amounts of information between parts of the program. The training data, parsed html and element lines all had to be stored in files and read into each program where they were needed. This challenged me to develop a robust directory architecture inside my system in order to maintain the pipeline.