Movie Script Shot Lister

A Project Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment of the

Requirements of the Degree

Master of Science

By

David Robert Smith

May 2016

The Designated Project Committee Approves the Writing Project Titled

Movie Script Shot Lister

by

David Robert Smith

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

May 2016

Dr. Chris Pollett	Department of Computer Science
Dr. Jon Pearce	Department of Computer Science
Dr. Teng Moh	Department of Computer Science

Abstract

The making of a motion picture almost always starts with the script, the written version of a story envisioned within the mind of its creator. The script is then broken down into shots. Each individual shot is filmed and then they are edited together to create the motion picture. The goal of the Movie Script Shot Lister thesis project is to be able to read in a script for a movie or television show, and automatically generate a shot list. While a script is text, a shot list is the blue print for how to visualize that script, so the shot lister tool will need some sort of information about the visuals and how they correspond to text that is written in the script. The basis of the lister is an application of artificial intelligence, a supervised learning algorithm to generate the shot list. First, training sets are generated by manually marking up scripts with shots. These training sets are fed into the program in order to give it a basic understanding of how to choose shots. The thesis project is composed of a few basic parts:

- a **parser** for procedurally breaking down a script into its components, which the rest of the program can use and understand;
- a **lining tool** for creating the training sets as well being a viewer for the final output;
- a **vector populator tool** for processing the training sets and storing the data;
- the **lister tool** itself which outputs a lined script;
- and a **comparer tool** for comparing different outputs.

Acknowledgements

I would like to thank my advisor, Dr. Chris Pollett for all his advice, collaboration and guidance with this project. I also appreciate the assistance of my committee members Dr. Jon Pearce and Dr. Teng Moh who gave me a lot of inspiration for creating this project.

I would like to thank my father, Dr. Robert L. Smith, Jr., for his countless hours of support on this project, as well as Ms. Kyra who provided encouragement along the way.

I am also grateful to all my friends and family who provided testing for this project and the rest who believed in me.

Table of Contents

Chapter: Introduction

During the early phases of making a motion picture, writing a **script** that will lay the framework of the complete motion picture is usually one of the first steps taken. (Although, in often less successful ventures, writing the script takes place after actors are hired and sets are built). The writer envisions locations, characters, props, dialogue. She builds a story structure that she hopes will be filmed. Once the story is written, it is usually passed off to a director or producer who begins the next phase of pre-production. Pre-production includes all kinds of things from building sets to hiring actors and crew. It also includes the very important step of taking the script that the writer has written and breaking it down into a **shot list**. This is a list of shots that take the written word in chunks and bring them to the screen. The shot list is often also drawn into pictures called storyboards, but even without the drawings, that first step of creating what exactly to film is very important. Describing a shot can be done with just words as long as the description paints a picture in one's mind. The shot list denotes when a new shot should begin and when an old shot should end in the form of a **cut**. It specifies what type of shot each shot should be from **close up** to **wide shot**. It should specify how many characters should be in a shot. It may even specify what kind of camera **motion** might be wanted for a shot.

Typically the shot list is created as a collaboration between the director and the cinematographer also known as the director of photography. Together, they break down the script based on emotional beats or actions. They attempt to decide the best types of shots to show off each moment in the most appropriate way possible. They want to use camera work

and cutting to emphasize and/or define subtext and show the deeper meaning of the script. There are no right or wrong ways to create a shot list, but different ways can give different meanings and even represent a very different movie. Sometimes, you can even identify a cinematographer or director by nothing other than the shots themselves.

The idea behind my project is to have a computer use **artificial intelligence** to take a script and create a shot list of its own. This will be done using **supervised learning**. That means taking scripts that already have shot lists and using them as a basis for the algorithm when deciding shot lists for scripts for which the program does not have shot lists yet. These scripts with shot lists already attached are called **training sets**.

There are many different artificial intelligence algorithms. For this project, **Naïve Bayes** was chosen. It is known to not typically have the "best" outcomes, but when it comes to creating a shot list, it is very difficult if not impossible to recognize if the output is actually the best as that is a matter of opinion better left to film critics. It is even difficult to judge if the output is "correct" or viable as there are a virtually infinite number of interpretations for a given script. We shall look at the outputs more as we get to the end of this paper.

To begin, it will help to look at what others might have done in the realm of artificial intelligence related to art and data collection.

Chapter: Previous Works

In researching this topic, I was unable to find any project, research paper or piece of software that matches or attempts to do what this project aims to attempt. That is, specifically, using artificial intelligence to create a shot list based on a film script. I found an article about using artificial intelligence to help create a script with the intention of making a hit (http://www.nytimes.com/2013/05/06/business/media/solving-equation-of-a-hit-film-script-with-data.html), but not breaking down a script into shots. I found many fine resources on how a human might go about breaking down a script into shots (a really good one here: http://nofilmschool.com/2013/10/cinematographers-process-part-1-breaking-down-the-script), but not a computer. There is also a recent proliferation of mobile apps providing tools for recording, scheduling, and using the shot list (http://www.shotlister.com/, https://itunes.apple.com/us/app/shotlist-movie-shoot-planning/id424885833?mt=8, http://nofilmschool.com/2012/10/diagram-shotlist-and-pocket-block-with-the-shot-designer-app, http://www.hollywoodcamerawork.com/sd_index.html). Film making, at least creating shot lists, and artificial intelligence are two concepts that do not seem to intertwine currently, it would seem. This could be indicative that it cannot be done, or it could be that no one with a background like myself, both in filmmaking and computer science has ever attempted it. I discussed this project in detail with various professional film makers (who are not computer scientists), and it was exactly their belief and conclusion that it could not be done. One even angrily pointed out that it shouldn't be done. He said, how on Earth could a computer ever be able to understand the nuances of what is going on in a scene?

Indeed, it is difficult to imagine a computer coming up with any piece of art from a painting to a poem, to a song, to even something like a shot list from out of nothing. Sure, a computer could randomly generate a bunch of numbers that produce something new. It is also said that if you leave monkeys in a room with typewriters long enough, they will type out the complete works of Shakespeare. A computer could also randomly generate a shot list, but it wouldn't be very good. So, this project does not attempt to spontaneously generate art out of nothing. The shot list comes from something, more than just the words written in the script. It wants to procedurally transform the creativity of others into something new. It wants to take the shot lists of others and extract characteristics from that during the training phase to apply to a new script. Now, this is certainly something we've all seen done before.

For example, a program like Photoshop can take a photograph and apply a filter to automatically make it look like a painting. You could probably even specify what artist you want the painting to look like. In a paper entitled "A Neural Algorithm of Artistic Style" (Gatys, Ecker, & Bethge, 2015), researchers at the University of Tubingen in Germany did a project exactly like that. You give their program a picture, specify the artist from a list they generated and it will render out a painting, and in the style of that artist. This isn't exactly what I am doing, but it shares some elements. What they do is achieved by studying the patterns of the various artists and capturing the elements mathematically. They then apply this math to the photograph to create an artistic output. In its essence, my project will be doing the same type of thing. Although the way I collect, process and apply the data will be different. You give my program

the script as its input, along with the artistic essence derived from the movies of previous filmmakers which are combined together to create an artistic output.

My project uses humans as the data collectors, who watch the movie that corresponds to a particular script and apply written notes to its script in a systematic way based on what they artistically discover in the movie. Although my Shot Liner tool used to make these notes is different, applying qualities to text has been done before for a long time with **qualitative data analysis**. Examples of software that do this include MAXQDA (https://en.wikipedia.org/wiki/MAXQDA) or Nvivo (https://en.wikipedia.org/wiki/NVivo). I chose to write my own software because I can control exactly how the data is going in and how it is coming out.

The relevant previous work which is most common is actually many examples and applications in the literature of Naïve Bayes. I could note any number of pre-existing papers on the subject, however, I think the most relevant is the one from the Artificial Intelligence textbook I used at San Jose State University, Artificial Intelligence a Modern Approach (Russell & Norvig, 2010). Most examples are simple, and what I am attempting to do is actually fairly complex. Most examples have a binary output. I actually seek several outputs, and only one of them is binary. The rest have several possible outputs. We will get more into how all this works later.

Before we get into discussing the algorithms and the more technical aspects of the project, it helps to go back and get a little bit of history about film making and the components that the lister tool will be working with during the course of the project.

Chapter: Of Scripts and Shot Lists

To help understand what this project is trying to achieve, it helps to start with a basic understanding of both how scripts and shot lists work and the components they are made of or built upon. Writing scripts, also known as **screenplays**, is an art and a creative process. Nevertheless, best practices for screenplay writing, also called **scriptwriting** or **screenwriting**, includes following the established standards for screenplay structure. This structure is taught in film-school and generally mimicked by people teaching themselves to write screenplays. There even exists industry standard screenplay writing software that thankfully enforces the established structure. Like any other profession, it is beneficial to follow the standards set up by the industry. This is beneficial to us in this project, because it means that our program can follow this structure too when procedurally breaking down a script. The guidelines are used by the parser to break down the script into classes and fields which are easily turned into JSON files for storing and furthering processing.

In its most basic form, a script is just a series of **scenes**. Each scene usually takes place in a single **location** and can be of variable length. Scripts have **characters** and other objects like **props** which each appear in one or more scenes. Each scene starts with an easily recognizable header. This is followed by a series of text blocks which I have called **action blocks** and **dialogue**

blocks. All blocks are separated by **blank lines**. An action block describes what is happening in the scene. It may tell us which characters are there and what they are doing. It could describe scenery, or props. It may even tell us what the camera is doing. Dialogue blocks share with us what the characters are actually saying in a scene.

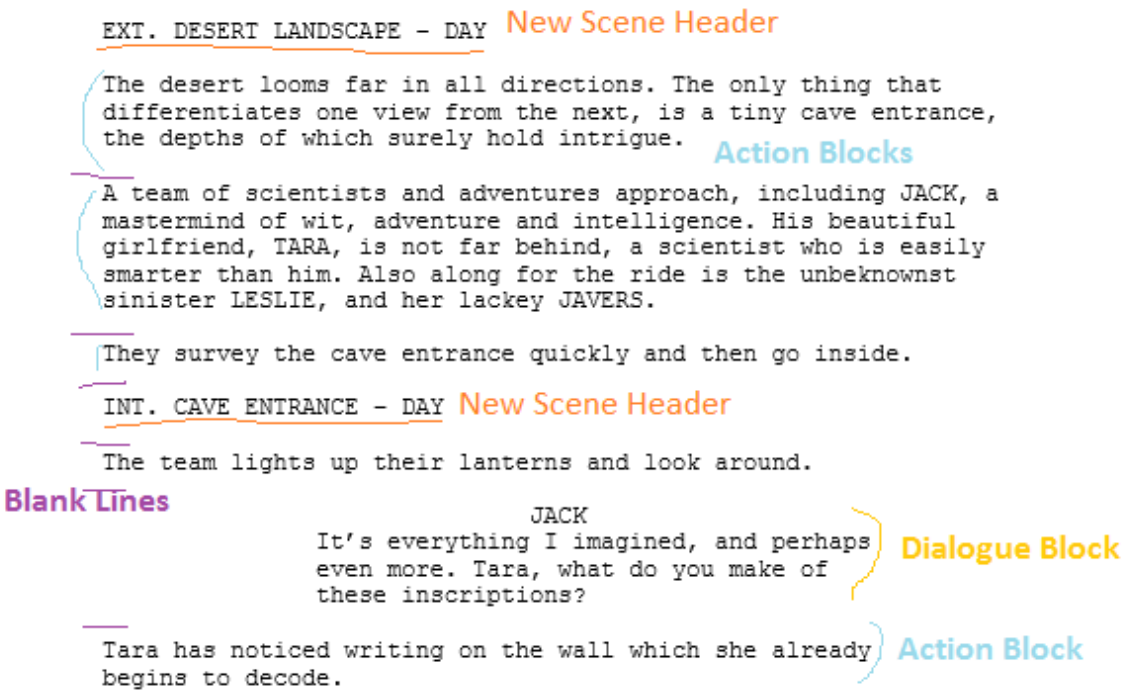


Figure 1 Script Breakdown

Scene headers in most cases follow a fairly strict format. Here is one sample:

"EXT. TRAIN STATION – DAY"

They are supposed to be all upper case letters. (Although, in some cases, a stray lower case letter may appear and the parser has to take this into consideration). They always start with a binary marker stating that the scene is **interior** or **exterior**. The most common markers are "EXT." or "INT." but these can vary somewhat. This is followed by the actual location, like "SAMMY'S BEDROOM", "MOVIE AUDITORIUM" or "PARK". Finally, you have a marker indicating

the time of day, such as “DAY” or “NIGHT” but it could something like “EVENING”, “DAWN” or “DUSK”. Some scripts or scenes might leave this off if it is not important. It is also pretty common that the scene number will be written at the front of the header, or at the beginning and end.

Action blocks do not necessarily follow any sort of standard, although it is generally accepted that important things are written in all upper caps. For example, the first time a character is introduced, good screenwriting practice has the character name all written in uppercase with subsequent references written normally. People, just like parsing programs, like to have markers so they can make mental notes. Props when introduced are typically uppercase also. Sounds and scenery may be uppercase. Instructions for camera motion are almost always uppercase. In some cases, important actions the characters take will be uppercase. It can be anything the writer deems important enough to bring special attention to the reader, or the person creating a shot list.

Dialogue blocks, like new scene headers also tend to have a pretty rigid formatting. Although, it is not just contained all on one line like a scene header. First of all, they always start after a blank line. Next, the first line is always written in all uppercase. In fact, this first line is the name of the character speaking the dialogue. i.e., DARTH VADER. The next line must not be a blank line and can be written normally in upper/lower case words and letters. You may also have descriptions in parentheses, both on the line with the character name as well as sprinkled throughout the dialogue. A common thing to see in parentheses is (O.S.) which means off-

screen or (V.O.) which means voice-over. But it can be lots of things including mood, direction or even actions the character is doing. The dialogue block ends when it is followed by a blank line. Unlike action blocks, dialogue blocks are also supposed to be indented a certain amount typically with a certain number of tabs. The number of spaces is not always standard, and as it turns out, counting these tabs or spaces wasn't necessary information for the parser to have.

There is much more subtlety to writing a script, but these are the basic fundamentals which allow us to move forward. With this greater knowledge of what scripts are made of, and their standard formatting, it is possible to begin writing the program that breaks them down, i.e., the parser.

The overall output of this project is a shot list based on the input script. A shot list is the series of shots that bring the screenplay to the visual medium. Each shot covers a certain portion of the script. It may be something like a close-up or a wide-shot. It may have just one character in it. It could be two or more. These characters could be standing next to each other or at different depths. The camera could be static or moving in many different ways.

While a movie is being planned, they actually typically plan several different alternative shots which could be used for each line of script. This is called **coverage**. Sometimes you do not know until you reach the editing room exactly which angle is the best for any given moment. Although, many different shots may be filmed, when the film is complete, there is exactly one shot covering any given moment, and every moment has exactly one shot. For practical

reasons, this project takes the point of view of creating a shot list of a finished film, not a shot list to use for filming, although it could be adapted in future iterations. Given this condition, what we are really after is figuring out when a cut takes place in the script.

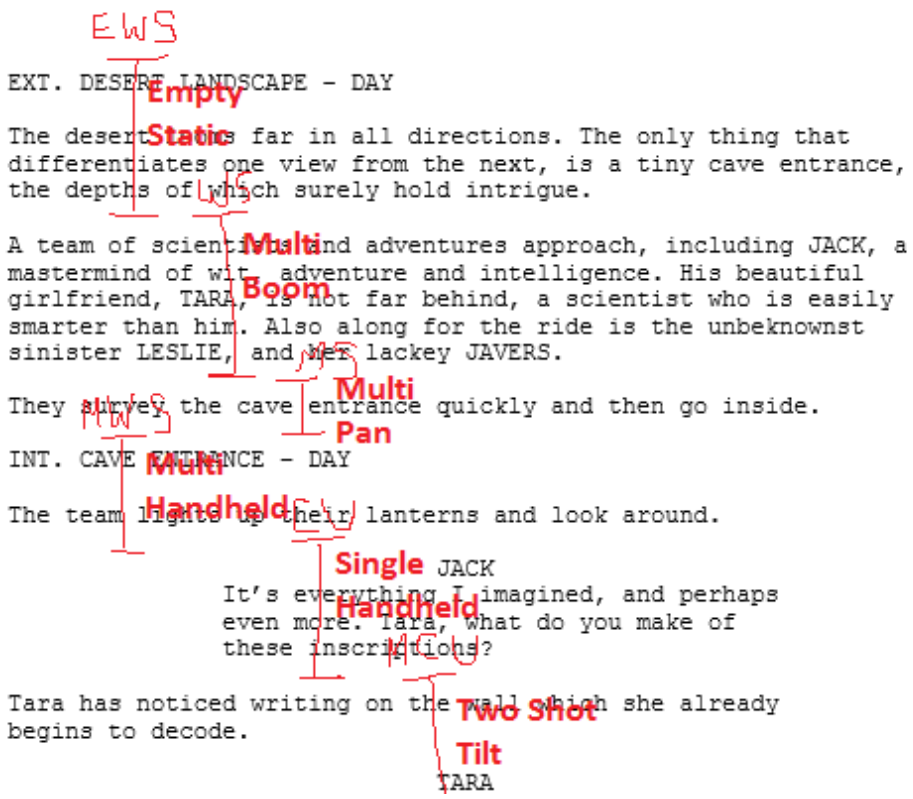


Figure 2 Lined Script

A **cut** is a moment when one shot ends and another begins. This name was presumably given because while editing actual film stock, the physical film was cut at the beginning and end of the shot and then the shots were spliced together with something akin to scotch tape. So, once we mark where the cuts are on the script, we know where the shots are. Then, we just need to figure out what the content of the shot should be given our other conditions which we call shot type, clean type and motion. This makes up an entire shot. In the context of this project, I am calling these the **target features**. Here are the options for the target features:

Cut	ShotType	CleanType	Motion
NoCut Cut	ECU – Extreme Close Up CU – Close Up MCU – Medium Close Up MS – Medium Shot MWS – Medium Wide Shot WS – Wide Shot VWS – Very Wide Shot EWS – Extreme Wide Shot None Other	Single – 1 person SingleOTS – Single over the shoulder Two shot – 2 people Multi – more than 2 people MultiOTS – multiple over the shoulder Empty – nobody there	Static – camera doesn't move Loose – camera moving a little Tilt – tilting up and down Pan – panning side to side Zoom – lens in and out Dolly – camera moves Steadicam – camera carried Crane – camera booms up/down Aerial – camera in air Handheld – held in hands DutchTilt – side rotate DollyZoom – zoom and dolly Circle – circler around Other

So, with the broken down script, and a sense of what shots are, we can begin to create the necessary programs and algorithms which together will make the Shot Lister tool.

Chapter: The Parser

The first tool created for this project was the **parser**. Parsing the raw script is the first step in creating a shot list, however, the parser itself does not provide the decision making logic for picking the shots. The parser is automatic. It is not interested in creating the shot list, only in breaking down the script into a data structure that can then be used by other programs and components of the overall project.

As all parsers do, it reads in plain text and based on the content, breaks it down into pieces and categorizes them. This parser has been programmed with the rules of how a screenplay is supposed to be structured. It categorizes several things at the same time. It keeps a list of both scenes as well as **objects** that may appear in different scenes. It is mainly interested in objects

that are characters, but it will pick out anything that is deemed in important. As such, the main data structure is called “script” and it contains lists of data structures called “scene” and “scriptobject”. It starts by reading one line at a time, but as it discovers different things, it will consider reading in several lines and process them all at once.

The parser knows what the header of a new scene is supposed to look like. Different scripts have different variations, but as many variations as could be found have been included into its logic. (I downloaded many different scripts during the course of this project, and used them to discover these sort of variations.) As mentioned earlier, the header should have an interior or exterior, the location itself and a time of day. Once the parser has found one of these headers, it will create a new scene data structure and add it to the script structure. It then makes this new scene the current scene. Any previous scene will be considered finished for now.

Then, it will continue down, checking a line, figuring out what it is.

There are many blank lines in a script, and these actually give useful information. For example, a header is supposed to have a blank line before and after it. Action blocks and dialogue blocks also are supposed to have blank lines before and after them. And later on, when the parser is running the artificial intelligence algorithm, knowing about blank lines is useful too, so the parser records all of them.

Dialogue blocks are a little harder to detect but there are still standards. The parser has been set up to look for three things when it comes to dialogue: a blank line followed by a line with all uppercase letters followed by a line which is not blank. Once a dialogue block has been discovered, it will actually take all the remaining lines of the block until another blank line is found and concatenate them into one string which represents the dialogue. Another important thing happens when a dialogue block is discovered. It means that a character has been discovered too. This character is both added to the list of characters in the overall script data structure as well as added to the list of characters that appear in the current scene. If the character already exists in either of these lists, it is of course, not added twice, although new references may be added. This character is also ultimately linked to the created dialogue block. The scene itself keeps a list of all its own objects such as blank lines, dialogue blocks and action blocks.

If the parser cannot identify a block of text as any of the above, such as new scene header, dialogue block or a blank line, it assumes it is an action block, concatenates all the following text until a blank line is reached, and labels it as an action block. This is not quite the final step, however. The action block is scanned for words that are all uppercase. These are considered to be important objects. They may be characters, sounds, camera directions, actions, etc. The parser may later identify them as characters, which it will keep track of.

When the parser finishes parsing the script once, it scans through it again, and checks to see if any characters it identified later on were not included in scenes that mention them in their

action blocks. If it finds them, it will include them during this stage. This is pretty rare, but there have been some cases.

In addition to creating the script structure, the parser also creates stubs for the placement of the final output of the lister tool, once it has completed its decision. Specifically, it stores each line of script, along with a Boolean operator to represent whether a cut takes place on that line or not. It also has an integer field for each shot type, clean type and motion type. Each integer represents a respective string. It made more sense to store them as integers to save space and make referencing easier. In a similar way, objects in a scene are stored as numbers, which all can be looked up in the main script data structure.

The parser populates the stubs with default data. The default for a cut is to say there is a cut for a new scene, and otherwise no cut. It defaults to medium shot for shot type, single for clean type and static for motion. These values are what you will see when you open a fresh script, and on average, they may be about right, but we do later need to be careful not to include them in lister algorithm.

With this, the parser's work is complete. The output it has created will be used both by the liner described in the next chapter, as well as the lister tool itself, when it is ready to make decisions about cuts, shot types, etc.

Chapter: Liner

Figure 3 Liner Tool

The liner tool is in many ways the viewport into this project. It was the genesis of this project too, as it started with the idea of lining film scripts. Why is it called a liner? In film-making, one of the very important jobs is script supervising. The script supervisor carries a copy of the script around the set and marks on it with a pen the shots that have been filmed. She draws a vertical line down the script from the point the shot starts until the point the shot ends. She then labels it at the top of the line with the type of shot it is, such as medium shot, extreme wide shot, etc. If the shot changes, she will mark that at the point it changed. She can also add information about the type of motion or if it's clean. This process is called lining a script. In essence, the shot liner tool is a way of marking the script in a similar way, but with a program instead of a pen to physical script.

The liner produces a GUI representation of the script. Each page of the script is represented as a page on the liner tool. Each individual line of the script has info about it, including the scene number and what line it is overall. It has a checkbox to represent if a cut happens. It also has drop down boxes representing shot type, clean type and motion. To line a script, one merely uses these features.

There is also an object window in the liner tool which shows all of the script objects, particularly the characters. Of course, people aren't objects, but in a computer science sense, it makes sense to call them objects in order to categorize them. Each object has a drop down box next to it saying what it is such as character, prop, sound, action, etc. There is also a list of the scenes the object appears in.

The liner tool serves two main purposes. These are either to line a brand new script, or to view a script that has already been lined. This latter feature is very useful for either viewing a script that someone else has lined or for viewing the output of the lister tool itself which is described in more detail later. As such, the liner tool can be started in two ways, either to begin a new script, or load up a previously lined script. In actuality, it is not necessarily opening a completely lined script, but rather a script that is in the lined format.

Entering a new script takes a script in text format, and assumes it is properly formatted within the guidelines of screenplay writing as described earlier. It actually starts by running the parser.

The parser transforms it into a data format that the liner tool can then use. In fact, once it is parsed, it will look no different in format to the liner tool than an already lined script. It may assume that the parser's default values are in fact the correct values for the script.

The liner naturally allows saving the lined script. It saves it in whatever state it is, whether the script is fully lined, partially lined or even if it was just read in by the parser. This is beneficial as it allows lining the script in parts or as time allows. It can be saved, the liner tool closed, and then reloaded back up to continue work where it was left off. It has actually been programmed to save output in two different file formats. One is the native JSON. The entire data structure is actually serialized as a JSON. This is merely saved to an external JSON file. The other file option is a zip. The zip file is just the compressed JSON. Having it as a zip saves space. In fact, every program in this series will accept either a JSON or a zip, including the liner, the vector populator, lister and comparer tools.

There is a little bit more finesse to the liner tool that I will describe in detail. I added many hotkeys that allow quicker lining of scripts. It has tips and hints built in to the different features to guide the user along. There is a place to put an optional timecode to help keep your place.

With the liner tool complete, the next phase of the project was ready to begin: collecting training sets.

Chapter: Training Sets

All supervised artificial intelligence learning algorithms use training sets to make themselves "smarter" or at least capable of the task. The more training sets the tool has, in theory, the better the results it will output. Training sets can take on many forms depending on the application. In our case, a training set is a lined script.

Lined scripts com in a variety of forms. It can be a script lined by an expert movie maker who merely looked at the script and picked out the shots he felt most appropriately suited the story he saw unfolding before him. It could be an inexperienced film maker who just lined as he saw it. In fact, a script could be lined in countless different ways and still look like a valid film. It could also be lined by watching a finished movie and marking up the script exactly the same way as it appears in the film. For the most part, this is the approach taken on this project.

Several films were picked as candidates to be lined. First and foremost, a script had to be available for any chosen movie. Almost as importantly, it had to be properly formatted so that it could be read by the parser and loaded into the liner tool. I even took it one step further, and saved the parser output and loaded it back in again because in a few cases, a script had strange characters that couldn't be read back in. (I used these opportunities to update the tool).

After that, a movie was checked to see how closely the finished film actually matched the script. The scripts available were usually shooting scripts, meaning they were written before

making the movie. Not many scripts are available in proper format that are transcripts written after the movie was made. So, in many cases, the director just uses the script as a rough blue print, and ends up changing many things including dialogue and scene order. Sometimes characters are removed or their names are changed. In extreme cases, the entire plot might have drastically changed. It's normal for there to be little changes, and almost rare for a final movie to exactly match its shooting script. Still, I tried to come up with a selection that had a script that matched the movie as closely as possible.

Once a wide selection of approximately 100 suitable movies was complete, I picked the actual movies to use for the training sets based on the sort of output I might want, such as a type of film, a genre, or a shooting style.

I took it upon myself to line the scripts, with an initial intention of trying to get all 100 scripts lined by the end of the thesis project. The first script I lined was Ghostbusters, and it was quite an undertaking. I most certainly was not able to do it all in one sitting. It took me over a week of working on it in chunks at a time to finish lining this single script. I did not actually time how long it took, but my estimation was that it took between sixteen and twenty hours. I quickly realized that I was not going to be able to get a hundred scripts done in total. Certainly not by myself, and probably not even with help from other people.

I set out to find other people to help with the process. Several ideas were thrown out, including crowdsourcing, or paying people, but at the end of the day, neither seemed to be an effective

solution as it was just too time consuming. Not too many people were interested in devoting the amount of free time required to finish and it would have been too costly to pay people. In addition to myself, my father lined several scripts, and my girlfriend lined one too. A few other family members and friends agreed to line scripts, but once they tried, soon realized it was a daunting task. See Appendix XX for my father's description and thoughts on his time spent lining!

Lining the script, although extremely difficult and time consuming, turns out to be very interesting. I could probably write an entire paper myself on just that, but suffice to say, it is extremely interesting to see up close and personal the decisions a director, cinematographer, editor, etc. made when shooting and selecting the shot choices that they did. It is also with a certain sense of wonder that these lined scripts are ultimately fed into the lister program to see what sort of results it can come up with based on them.

As of this writing, ten scripts have been completely lined, with at least one more coming up. This is a far cry from the hundred originally hoped for, but given the amount of time and work required to get even that many, it comes with a certain sense of pride. And, as it turns out, it is still likely enough to come up with some interesting results. Keep in mind that even one script has thousands of lines. Each of those lines provides useful data to the training algorithm. Each script has dozens if not hundreds of scenes to provide good data.

With even just one script lined, the next step of the project was able to be started, constructing the algorithm for outputting a shot list for a brand new script.

The Naïve Bayes Algorithm: In two parts.

The Naïve Bayes Algorithm is actually executed in two parts that is two separate programs were written, each to do one task. They were written in tandem and use many of the same components and methods. The first program is the vector populator tool for creating the vector file. The second is the lister tool which takes the vector file and an unlined script and creates the final lined script as its output.

Chapter: The Vector Populator Tool

Naïve Bayes works by creating vectors from the training sets. What we ultimately desire is to know for each line whether we have a cut or not, what is the shot type, what is the clean type and what is the motion. Each of these four categories, I have mentioned we call the target features. And each target feature has a vector associated with it. The vector is basically a set of probability data. At its heart, what a vector is doing, is storing the conditions that took place during the various possible outcomes of a target feature. For example, it looks at all the times there was a cut, and records all the other events that took place on or around that line too while there was that cut. These other things that happened are called **features**. They are features of the script. One simple example of a feature might be, what scene object is a given line? Is it a scene header? Is it a blank line? Is it an action block or a dialogue block? What it is

will play into the calculation of the target feature. When you have enough different features, you start to be able to paint a picture of what the target features should or might be.

For sanity sake, I will use the word **option** in this document when referring to target features and **selection** when referring to feature. So the option picked for the cut target feature might be cut or not cut. The selection of the scene object feature might be an action block or dialogue block.

What does a vector look like exactly? It is called a vector because it is supposedly a one dimensional array of data. However, each component itself has an array or mapping. A vector for a target feature is populated with all the features. Like a target feature has at least two options of what it could be, a feature also has a selection of options. Like target features, they are predefined as well as capped. With scene object type, for example, we know it has to be a new scene, blank line, action block, dialogue block. So for each target feature, the array contains all features, each of which contains all the possible selections of that feature, and then each of these contain an array of all of the options of the original TargetFeature. If that sounds confusing, here is a programmatic representation:

targetFeature[].feature[].selection[].option[]

When the vector is first created. It is initialized with all values at zero. This is important because as we populate the vector we want to be able to continue increasing the counts by one.

So in order to actually populate the vector we start by scanning the script line by line. Keep in mind that it's not the same as scanning the script line by line the way the parser does because in this case, the parser has already processed the data, so it's in a form the vector populator can easily use to get the data it needs to put in the vector. And because it's already a marked training script, we already know about what each target feature is set to. And, we can see, calculate or keep track of the selected options for each feature depending on what it is. Now, as an example, say we hit a line that is new scene. And say that it has a cut. In the new scene (cut) vector, we select the scene object feature. In the selection of that feature, we select "new scene". And then under that selection, we pick the option of "cut". Finally, we reach an ordinary integer counter. We increase this number by one.

There are many different features, each of which has its own way of calculating its value. Some are simple like the scene object type which was calculated by the parsing and stored in the lined script. Things like the scene number or the number of characters in the scene are similarly pretty simple to come up with. Others begin to become a bit trickier, like measuring the number of lines that have passed since something happened, like maybe a new scene or a cut.

For my own sanity, I came up with a concept of two different kinds of features. One is pure and the other is non-pure. A pure feature is any feature that you can glean from the parsed script itself regardless of the lining of the script or the output of the lister tool. A non-pure feature is one that is based somehow on the target features, or may even be a target feature itself. (All

four target features are also stored as features as well). Calculating something like how many lines have passed since a cut is non-pure because it is dependent on the last time a cut was calculated. If the tool decided the previous line was a cut, only one line has passed, whereas if it decided on one twenty lines earlier, that may change the outcome. I mention this now, because later on when utilizing the vectors, we need to be mindful of the non-pure features.

Aside from some book-keeping with calculating the feature selections, it's basically that simple to populate vectors. We do have to keep track of a lot of numbers and be careful in doing so. Increasing just by one may not seem like a lot, but this will add up over time. For every line, every target feature must be considered. And for every target feature, every feature must be considered. So, right there, if we have four target features and twenty features, that's eighty variables we're manipulating just for one line.

The beauty of populating a vector is that you can have as much or as little data as you want. The more lines you use to train, the richer it will become. And, you can mix and match any training sets that have been created. For convenience, I created the shell of the vector populator tool as a GUI much like the liner tool. It has a file system which allows selection of the various training sets. It checks to make sure they are valid training sets, and it will also make sure a particular training set is only added once. Finally, it allows you to save the vector file with a custom name. This will allow creating different vector files with different combinations of training sets in order to compare their outputs from the lister tool. We'll discuss that more in the experiments and results chapters towards the end.

Chapter: Features

In this section we will list out all the features that are used in the vector and lister tool and briefly what they mean. It is important to note that the order of the features does matter, at least for the ones that are also target features, as what each target feature is determined to be affects what might be selected for the next features.

First the non-pure features:

- cut
    - One of the target features. Simply, describes if there is a cut or not on this line.
    - Whether there is a cut or not should play heavily into deciding the other target features. For example, no cut means it is more likely that a shot type would remain the same as the last line, however, a cut can mean shot type could be anything.
- shotType
    - The shot type that has been determined for the line.
    - This is an important consideration for clean type. For example, a wide shot may be more indicative of say “multi” whereas close up may be indicative of “single”.
- cleanType
    - The clean type determined for the line.
- motions
    - The motion determined for the line.

- linesSinceCut
    - A counter that keeps track of how many lines of script have passed since the last time a cut was determined.
- linesSinceShotTypeChange
    - A counter that keeps track of how many lines have passed since there was a change in shot type.
- lastShotTypeNoCut
    - The shot type chosen on the last line.
    - This feature is only included when there is no cut on the current line.
- lastShotTypeWhenCut
    - The shot type chosen on the last line only when there is a cut on the current line.
- linesSinceCleanChange
    - A counter for the number of lines since clean type changed.
- lastCleanType
    - The clean type that was chosen on the last line.
- linesSinceMotionChange
    - A counter for the number of lines since motion changed.
- lastMotionType
    - The motion that was chosen for the last line.
- tiltCount
    - A counter for how many lines have passed while tilt was selected for motion.

    - Tilt should not happen too many times so this feature is in place primarily to prevent that.
- zoomCount
    - A counter for how many lines have passed while zoom was selected.
    - Like tilt, zoom should be limited.
- panCount
    - A counter for how many lines have passed while pan was selected.
    - Like tilt and zoom, pan should be limited.

Non-Pure features added after separating each Target Feature into its own iteration:

- dialogueCountInShot
    - the number of dialogue blocks in the shot of the current line
- uniqueDialogueCountInShot
    - the number of unique dialogue blocks, not counting a character more than once
- actionCountInShot
    - the number of action blocks in the shot
- lineCountInShot
    - the length of a shot in lines

The pure features:

- sceneObjectType
    - This feature checks to see what the scene object type is on the line.
    - This may be a new scene header, a dialogue block or action block.

- linesSinceObjectChange
    - A counter for how many lines passed since the last time the scene object changed.
- linesSinceNewScene
    - A counter for how many lines have passed since the scene started.
- intExt
    - Returns whether the current scene is either interior or exterior.
- scriptObjectsInScene
    - The count for the number of script objects that have been found in the current scene.
- actionBlocksInScene
    - The total number of action blocks in the scene.
- dialogueBlocksInScene
    - The total number of dialogue blocks in the scene.
- sceneLength
    - the length of the scene in lines

Chapter: Lister Tool

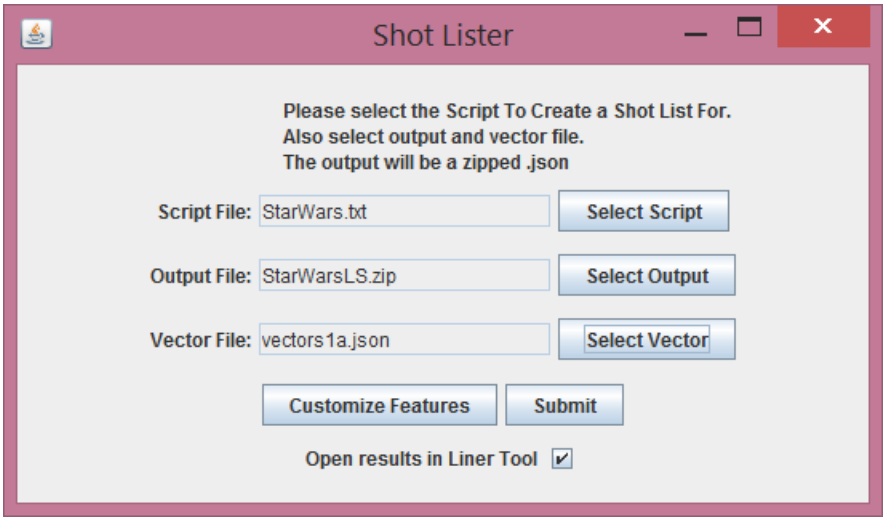


Figure 4 Lister Tool

The lister tool is in many ways the culmination of the entire project. It is what creates the lined script output. The lister tool takes two files as input, a vectors file as created by the vector populator tool and a raw unlined script just like the liner tool starts with.

The lister tool uses the parser the same way that the liner tool does. In fact, it creates a file that starts out exactly the same as what the parser hands the liner tool. It has the same data structure, with all the same stubs. This makes sense, because it is outputting a file that is the same. The difference begins here. Instead of a GUI which can be used to make the selections of the target features, like the liner tool, the lister tool now dives into the Naïve Bayes algorithm in order to come up with the shot list.

The Naïve Bayes algorithm of course uses the inputted vector file in order to come up with the shot list. This is where the ideas of probability come in to play. The Naïve Bayes algorithm is based off the idea of **Bayes Rule** (or Bayes Theorem). This rule is written like this:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

What this means is that the probability of A given that B happens, is equal to the probability of B given that A happens times the probability of A divided by the probability of B. Basically, if we have any three of these probabilities, we can calculate the fourth.

Here's a very simple example that applies to the data we already have. What if we know that a given line is a new scene? We want to find out what is the probability of a cut given that it is a new scene. We know the probability of a cut over all. From the training sets, the total number of cuts over the total number of lines is the probability of a cut. We also know the probability of a new scene. That is the total number of lines with a new scene over the total number of lines. We also had counted up all the number of times there was a new scene when there was a cut. This number divided by the total number of cuts gives us the probability of a new scene given a cut. We can put all that together to find the probability of a cut given a new scene using Bayes Rule. But, how does this help us decide whether we should have a cut or not?

This is where it gets interesting. We do the same calculation but instead of cut, we do it for not cut. So, we figure out, what is the probability of no cut, given a new scene. We find the

variables the same as before, and we will get a number. Now, we have come up with two probabilities. One is the probability of a cut given new scene and the other is the probability of no cut given a new scene. Now, I will tell you from my experience that there were on very rare occasions times when there was no cut on a new scene, but almost always, there was a cut on a line that was a new scene. This makes sense because a new scene almost always switches to a new location, and unless you do an intricate camera movement or have your characters cross over in a unique way, you have to cut to get there. So, with almost all training sets, you will get a result that the probability of a cut on a new scene is higher.

That, in a very basic way, is what Naïve Bayes does. It calculates the probability for each option of the target feature. It goes with the probability that is the highest. It's easier for cut, because there are only two options, cut or no cut. For others like shot type, it has to calculate the probability for each of the possible options.

And of course, there isn't just one feature as in our example. We have to use all of the probabilities for each feature, given the feature selection chosen. This is where it expands from Bayes Rule to the full Naïve Bayes algorithm. It is written like this:

$$Option = \underset{k \in \{1,\ldots,K\}}{\operatorname{argmax}} p(C_k) \prod_{i=1}^{n} p(x_i|C_k)$$

In English, that means we're taking the max of each of the options from 1 to K, as indicated above. Each option's probability is calculated by multiplying the probability of the selection given k times all the probabilities of each option given that selection. If you have a keen eye, you've probably noticed we dropped the probability of each option itself which would be in the divisor in Bayes rule. Since this same value would be in the divisor in all the products, i.e. for both the calculation of cut and not cut, it doesn't change how the outputs compare, so the extra calculation can be dropped.

That in essence is how Naïve Bayes works. The vectors have collected all the probabilities. So, now, much like the way the vector populator had to go down each line, the lister tool goes line by line too. It doesn't know the option for the target features yet, but just like the vector populator it can figure out which selection has been selected for each feature. It uses the vector like a look up table, plugging in the selections, and the probabilities pop out for each of those options given those selections. The product of all those probabilities gives the total probability and the comparisons can be made. With that, the option is filled in for each target feature on the line.

There are some known issues with Naïve Bayes. Probably the most well-known is that if there are zero occurrences of a picked option in a selected feature then you will have a probability of zero. Zero multiplied by any number is zero which would skew the entire results. To prevent this, some sort of smoothing needs to take place. Laplace smoothing should probably work. In

theory, we also want all the features to be independent, but it does not necessarily matter too much.

That is basically it, but a little bit of care does have to be taken about how the features are examined and even which features should be considered when calculating the product for final probabilities. The next chapter goes more into feature settings.

Figure 5 Feature Settings Form

Not satisfied to just use the feature probabilities at their face value, I went a few steps further and added customization for the settings. First and foremost, I added the concept of skipped features. These are features that I specifically don't want used in the product to calculate the overall probability. Secondly, I added the ability to give weights to the features.

There are definitely some cases where you don't want a feature being included in the calculation of a target feature. The most important case is when the target feature is a feature itself. Indeed, all of the target features are also features, and in some cases, this is appropriate. It is not appropriate however, to have a target feature use itself in a calculation. The reason this is a problem is twofold. Let us consider the simple cut as an example. If the feature is cut and the target feature is cut then you have selections of cut and no cut as well as options of cut and no cut, respectively. What will happen though, is that every time feature is cut, target feature will also be cut. And similarly for no cut. This could end up skewing values greatly. The probability of a cut when there is no cut is effectively 0 percent. You don't want to multiply zero times other probabilities because you will most certainly end up with zero. For this reason, it is important to skip a feature when it matches the target feature.

There are certainly cases where you would want to use a target feature as a feature though, and that's why they are included. For example, when deciding on what the shot type might be, whether it was a cut or not on that exact same line is actually pretty important. As a matter of fact, if there is no cut, the likelihood that the shot type would be the same on this line as the last line is pretty high, since you would expect that unless the camera moved or a character moved, things that do happen but not necessarily frequently, then the shot would be the same. On the other hand, if there is a cut, it might still be the same shot type on this line, like if you were cutting back and forth between two close-ups, but it is also just as likely that it would be a different shot type. Likewise, once you know the shot type, you might want to use that for the clean type. These are reasons why target features are included as features. Furthermore, each

target feature on each line is calculated completely before moving on to the next, so that it can purposely be used in calculating the next.

The order of calculating target features has been decided on as cut, then shot type, then clean type and finally motion, each calculation being included in the next. This order was decided starting with what instinctually feels like the most independent down to the most dependent. Meaning, you would really want to know if it's a new shot before trying to decide what the shot should consist of.

Another example of needing to include a skip feature is with features like "lastShotTypeWithCut" or "lastShotTypeNoCut". If there actually isn't a cut, you'd want to skip the first one. If there is a cut, you would want to not include the no cut variation. The last shot type to the new one as inferred above is very different when there is a cut or not which is why they were separated and skip feature is used here to make sure each one doesn't skew the other's data and input.

The skip feature concept is strictly enforced by the program and hard coded in. But variations on features' impact on the output are also soft coded with the introduction of feature settings. The feature settings are an extension of the Lister tool. Like the training scripts, and the vectors, they are saved into their own specially formatted file of type JSON. They have their own GUI window which is brought up from the lister where the settings can be changed, loaded or

saved. They generally work by manipulating the impact each feature has on the output calculations.

Each target feature contains its own settings for all of the features. So, given there are four target features, each feature actually appears four times. But each feature figures into calculating each of the four target features differently, and this is the reason they are each shown separately. Each feature is associated with two values, a Boolean and a double. The Boolean indicates whether the feature should be used at all in calculating the target feature. This is just like the skip feature mentioned above except it is soft instead of hard coded. This means it can be changed easily which is extremely useful during testing to see which features ultimately should be included in the calculations of each target feature.

The double represents the weight that the feature should have. The default is 1.0 which means it will be used normally. Keep in mind that each feature represents a probability that is being multiplied with other probabilities, so the weights are not actually being multiplied times each probability like as if you were trying to weight an average. Multiplication is commutative so multiplying one of the values by a number would be no different than multiplying any of the numbers by the same value. Instead, the probability is raised to the power of the weight. Keeping this in mind, a number lower than 1 will cause the probability to have less impact. Any number to the power of 0 equals 1, so a weight of 0 is the same as not using that feature at all. A number greater than 1 for weight will cause the probability of that particular feature to have a much higher weight.

It is important to note that regardless of how random the output appears or of what features are being used, the results are always deterministic. If the exact same raw script is entered with the exact same vector file with the exact same feature settings, the output will be the same every time.

Having the ability to manually change the settings of the features allows looking at the output to decide which features actually help in providing good results and which features seem to make it worse. In the next chapter, we dive into the refinement phase, where we actually pick which features to use, and what weighting to have on them.

Chapter: Refinement

Perhaps the biggest refinement I made was when I noticed there was a definite skewing of the results toward the first option of each Target Feature. As I mentioned in an earlier section what was happening was an issue with zero probabilities. If a newly inputted script has a selection of a feature that never occurred in any of the training sets, it will produce a probability of zero. For example, say the linesSinceCut feature comes up with a case where there's a 100 lines since a cut while running the Naïve Bayes on a script, but in all training the training sets, there was always a cut by the 99th line. (In actuality, there were many times where there were no cuts for over 100 lines, but for this example, let us assume that we always had a cut by the 99th line.) In the vector, all the options for 100 or over, would have 0 cases for any of the options of this feature. So the probability then would be 0 that this selection happens given any option.

Multiplying this probability times all the other probabilities would cause it to go to 0. This is not what you really want. Imagine if for all the options, it went to 0. Instead of comparing them against each other to pick the one with highest value, it would just default to the first one.

To solve this issue, I implemented Laplace smoothing. That means taking the probability and adding a small number to it so that it is never as low as zero, although numbers that would have been zero will be very small. This is appropriate because it will make the overall probability go way down which it should since we have clearly seen from the vector set that this outcome is rare if not unique. What is recommended is to add a Laplace constant on the top of the probability, typically the number 1, which is what I did. Multiply this same constant times the number of possible outcomes for the target feature, that is the options, and add that number to the bottom. This will ensure that if you are creating probabilities and if they are all added up with their complementary probabilities, it would still equal a total of 1 as before. This smoothing will affect the ratios of the probabilities slightly, but it is a small price to pay for ensuring the zero issue is taken care of.

Another major refinement came with adjusting the features. The very early versions of the lister tool, only had a very limited number of features on which to train. Of course, it had the four target features as features. What became apparent, was as features were added, the results really started to change, sometimes for the better, and sometimes for the worse. What is desired is to find the exact right configuration of features, i.e. which should be used for which target features, and what sort of weighting should be used for each.

Ultimately, certain features may seem like a good idea to add into the program, but it may turn out, they not only do not help, they make the output worse. And some features will help, but only when the weighting is shifted.

The final improvement I made to the algorithm was actually separating the picking for each target feature into its own iteration. Originally, I went through the script line by line only once, picking all four target features as I scanned each line. I realized that I was actually better off finding doing them separately, each with its own complete pass. Since each target feature already scans through all the features individually on each line, separating them into different passes actually doesn't add much time. What it did allow was to get more complete data on one target feature before processing the next. This was particularly important with cuts. Calculating cuts first meant that when calculating shot type, etc., the program would know where the shot started, ended and its duration. This meant being able to add new features like counting the number of dialogue blocks in a shot. For example, if a shot has one dialogue block, it is more likely to be a close up and a single. Whereas, if it has several dialogue blocks it is more likely to be a wide shot and a multi. Of course, making this change meant that choosing the order of calculating target features was even more important, and earlier target features couldn't use any of the results of later ones. The previous way allowed cut to see what the last shot type was, and now that is not possible. However, overall this change made for a great improvement in the results.

With enough tinkering, we can get results that visually may look pretty good. This brings up an excellent point though, how do we know if the results are better or worse and this brings us to our next chapter.

Chapter: The Comparer

How do we know if the results the lister has produced are good, or to use a different idea, how do we know if they are correct? This is a somewhat difficult question to answer, because as we already stated, creating a shot list is a bit of a work of art, and good art is always a matter of taste and opinion. So, it depends completely on interpretation. Still, we can make some comparisons.

To test the output of a shot list generated by the lister tool, we can compare it to the same script that was lined by a human while watching the actual movie, which is what the training scripts are. Remember, the training script is in the exact same format as the lined script as outputted by the lister tool.

The simplest way to compare the two scripts is procedurally with a program that compares how closely, line by line the two scripts match. To start, I wrote a program called comparer that uses the idea of edit difference between the two scripts. Take for example, the cut edit difference.

The comparer tool goes down the first script line by line. If there is a cut on one line, it will check to see if there is a cut on the same line of the other script. If there is not a cut, it will

begin to scan the other script until it does find a cut. It will first count the number of lines up from there until it finds a cut and then count the number of lines down until it finds a cut. It will return the smaller of these two values as the edit difference for that particular cut, adding that to a total. Note that if they both have a cut on the same line, a zero is added to the total. It then continues down the first script, adding up all these discrepancies. Now, it's not good enough to just scan down one script and compare it to the other. Say that the first script has very few cuts while the second script has a cut on every single line. This will actually return a total of zero even though the scripts are actually very different. In order to thwart skewed results such as these, the comparer does the same process again but starts by scanning the second script line by line and comparing it to the first. By the time the scanning has been done twice, there will be two totals values and these can either be added or averaged together. I chose to average them. The lower the number, the closer the scripts match. If you compare the same script against itself, the number will be zero.

The other target features use a similar technique, although they are a little more complicated. It scans down line by line, but instead of looking to compare only lines that have a cut, it will actually compare every line to the corresponding line of the other script. So, for example, if the target feature is shot type, and on the given line, the shot type is close up, it will check to see if the other script is a close up on the same line. If it is not, it will scan both up and down on the other script until it finds a close-up, and report back the short distance whether up or down. This is added to a total. If the other script actually never has a close up, it will scan all the way to the beginning of the script, as well as to the end, and return the minimum of one of those

distances. Since it does a similar scan on every line, you can imagine that the difference number will grow quickly, and it does. The differences for shot type, clean type and motion are much larger than cut type. Cut edit difference comes back often in the thousands where the others may come back in the tens or hundreds of thousands.

The comparer tool returns the edit differences for the four target features separately, as well as one combined number.

Now, results like these are not very good on their own. If you get back a number around 3000 for cut difference that does not mean much unless you compare it against something else. So, you may want to compare several different lister outputs against the original training set. (Different outputs are a result of using vectors with different training sets, and/or changing the weight and use of the different features.) The script that comes out with a lower number could be considered closer to correct, or a better output. Although, not necessarily, as again, there can be a lot of variation among what is considered acceptable output. Still, this provides a starting point. Perhaps one of the biggest faults of the comparer tool is that you need at least three inputs for it to work. You need the training script, and two other outputs. Each output using either different training sets or feature settings. In this way, you can really only judge the other two outputs against each other, which is better and not truly against the training script itself. Of course, that is what we really want: judging how good the output is against the original training set. All we know is that a low number is good. So, the comparer is not really a complete metric. Using the comparer gives us some data, but we really need another method of

comparison to get better insight. And moreover, to compare output against the training set itself.

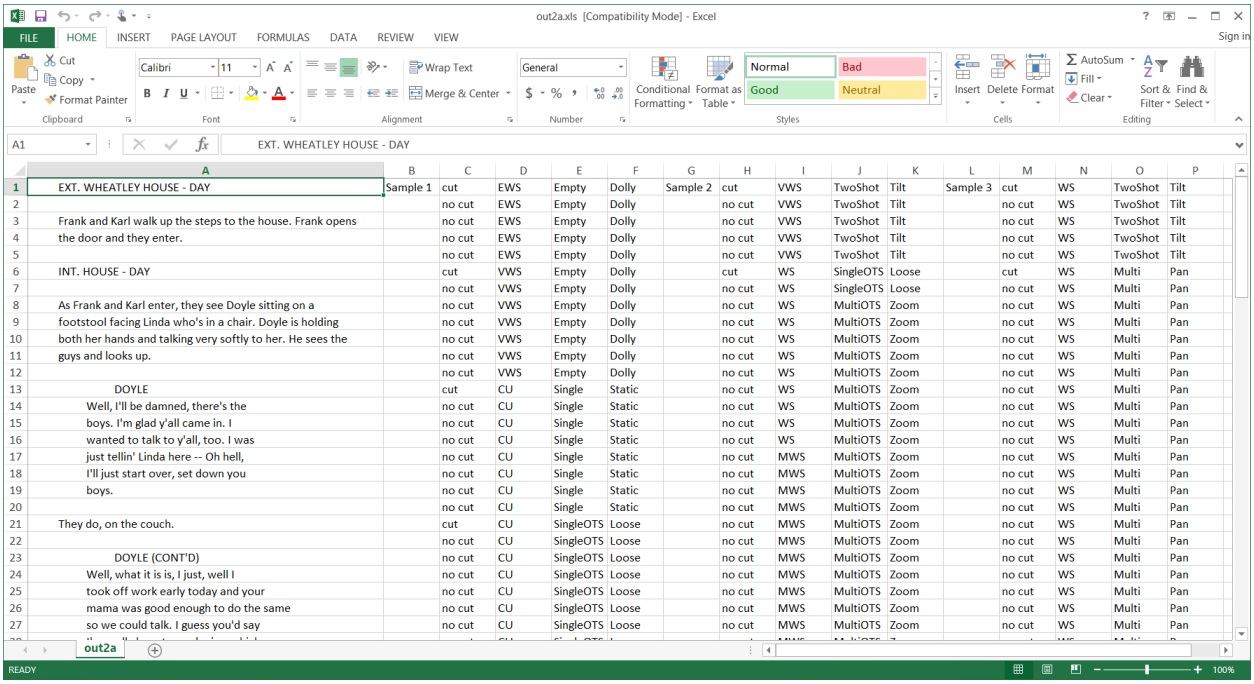
It is worth pointing out however that the comparer tool had other uses. There was more than one occasion that I wished to refactor the much of the code, both to tighten and clean up, while still desiring the exact same output. When I did this, I would output a lined script before doing the refactor and then output another lined script afterwards. I could then use the comparer tool to compare them. If it returned a 0, then I knew that my refactor successfully still outputted the same results despite all the changes in the code.

Chapter: Human Judgement

In addition to using the comparer tool to examine the outputs, it was decided that using human eyes was also a good idea. Humans can instinctually sense art better than a computer can. Of course, we did not want to overburden the generous humans who volunteered to examine the outputs, so, we decided to do only 100 line chunks of any given script.

In order to make it reasonably easy for humans to judge the material, I wrote another program called HumanJudge. It takes the lined scripts and creates a simple CSV file with the lines of script, as well as the picked options for the target features. It requires a minimum of two lined scripts, but will take three or more as well. Each line of the CSV file has the text of one line of the script, followed by the target features that were picked for that line according to each sample.

The HumanJudge program picks the sample randomly. So, it will likely grab a different set every time it is executed. It finds the length of the script, and selects a start line from 0 to the length of the script minus the sample size, which we decided would be 100. It then iterates through the same lines on all the input lined scripts and exports the options picked to the CSV file.



	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	EXT. WHEATLEY HOUSE - DAY	Sample 1	cut	EWS	Empty	Dolly	Sample 2	cut	VWS	TwoShot	Tilt	Sample 3	cut	WS	TwoShot	Tilt
2			no cut	EWS	Empty	Dolly		no cut	VWS	TwoShot	Tilt		no cut	WS	TwoShot	Tilt
3	Frank and Karl walk up the steps to the house. Frank opens		no cut	EWS	Empty	Dolly		no cut	VWS	TwoShot	Tilt		no cut	WS	TwoShot	Tilt
4	the door and they enter.		no cut	EWS	Empty	Dolly		no cut	VWS	TwoShot	Tilt		no cut	WS	TwoShot	Tilt
5			no cut	EWS	Empty	Dolly		no cut	VWS	TwoShot	Tilt		no cut	WS	TwoShot	Tilt
6	INT. HOUSE - DAY		cut	VWS	Empty	Dolly		cut	WS	SingleOTS	Loose		cut	WS	Multi	Pan
7			no cut	VWS	Empty	Dolly		no cut	WS	SingleOTS	Loose		no cut	WS	Multi	Pan
8	As Frank and Karl enter, they see Doyle sitting on a		no cut	VWS	Empty	Dolly		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
9	footstool facing Linda who's in a chair. Doyle is holding		no cut	VWS	Empty	Dolly		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
10	both her hands and talking very softly to her. He sees the		no cut	VWS	Empty	Dolly		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
11	guys and looks up.		no cut	VWS	Empty	Dolly		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
12			no cut	VWS	Empty	Dolly		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
13	DOYLE		cut	CU	Single	Static		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
14	Well, I'll be damned, there's the		no cut	CU	Single	Static		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
15	boys. I'm glad y'all came in. I		no cut	CU	Single	Static		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
16	wanted to talk to y'all, too. I was		no cut	CU	Single	Static		no cut	WS	MultiOTS	Zoom		no cut	WS	Multi	Pan
17	just tellin' Linda here -- Oh hell,		no cut	CU	Single	Static		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
18	I'll just start over, set down you		no cut	CU	Single	Static		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
19	boys.		no cut	CU	Single	Static		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
20			no cut	CU	Single	Static		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
21	They do, on the couch.		cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
22			no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
23	DOYLE (CONT'D)		no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
24	Well, what it is is, I just, well I		no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
25	took off work early today and your		no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
26	mama was good enough to do the same		no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan
27	so we could talk. I guess you'd say		no cut	CU	SingleOTS	Loose		no cut	MWS	MultiOTS	Zoom		no cut	WS	Multi	Pan

Figure 6 Human Judge Output

The HumanJudge takes something like a double blind approach. That means although I know which scripts I am feeding the HumanJudge program, it strips that information before presenting the data to the human subject. There are no references to the lined script file names in the CSV file or indicators to the human which script is which other than the content itself. Each sample is labelled “Sample 1”. “Sample 2”, etc. The program also actually randomizes the orders of the samples. So, if I were to insert the same three scripts, on different occasions, it

would output the samples in a different order. Then, when I pass the CSV file to the human subjects, even I do not know the order. The HumanJudge program also outputs a simple text file that has a legend stating which sample belongs to which lined script file. I do not send this along to the human. I just use it is as a reference to know which files are which after they send back their preferences about the samples.

The program runs on the assumption that the lined scripts being inputted are all actually the same script, albeit with different picked target features. It needs to go by this assumption because it is only printing the lines from the scripts once, which means it is assuming that the lines are the same in all the scripts. Without that, the data is useless because the second or third sample would be referencing some different lines of script and we would have no comparison. This assumption is also important because it is pulling the same datalines from both scripts. If the scripts are not the same, and do not have the same length, it may try to pull a line from the second script that is out of bounds and could throw an exception.

Now, the HumanJudge program also has a fault in that it is only sending a sample to the human judgers. To truly judge the output, it should be sending the entire script. In this way, the human could judge the output as a whole. Like, does output follow a style all the way through? Does it get the subtly of what is happening from one scene to the next? This would be nice, but like in any statistical analysis, we really only can afford to get a certain amount of data, which we

assume is representative of a larger which. In other words, we take what we can get and go from there.

With this program complete, it was relatively easy to create human judgement samples to send to people that they could quickly go over and get back to me with their findings. Of course, in order to do that, I first needed to come up with what the actual experiments would be.

Chapter: Experiments

In order to test the output of the lister program, several experiments were conducted. What follows are descriptions of the experiments. Some of the experiments only have two things to compare, the training set and the lister output. In these cases, the human subjects were asked to say which version they preferred or which version seemed more natural. Some of the experiments have three or more different outputs in which case, the human subject was asked to rank them from best to worst. Also in the cases of three or more, the comparer tool could be used as well.

I had over a dozen human judgers who were willing to participate. Below the description of the experiments are the results. I actually looked at the numbers first before I looked at the files to see if I could guess which numbers corresponded to the original training set. When I could not guess, I saw that as a positive. The human choosing the lister output over the training set is not necessarily the goal. The goal is that they are chosen an even amount, or at the very least the

training set is not always preferred. If the lister output is chosen about the half time, I concluded that the output is reasonable.

Experiment: Basic

The most basic experiment is to take all of the training sets except one, and use them as input into the vector populator. Then, take the script only version correlating to the unused training set and feed it into the lister tool. So the system will be trained with n-1 of the available training scripts and then will produce its shot list for the nth movie which can be compared with the training script produced by a human for that movie. How good of a job does it do? Naturally, a human may be interested in looking at the results. Does it appear to do a good job on visual inspection? What other ways can we examine its output?

Experiment: Control Sample

A couple of the scripts that were used to create training sets by watching the movie and closely lining the script to match the actions on the screen where also used to make control samples. The control sample gives a third sample to compare against the standard training set in addition to whatever output the liner tool might give. The idea is that a person will once again mark up the script, but this time, instead of watching the movie, they will only have the script, and they will use their own insight into the script and artistic judgement to line the script.

With that complete, the lister outputs its own version of the same script. Like above, we will probably want to use all the training sets as input except for this particular script. The lister output is compared to the original training set and the control sample is compared to the training set. The question is, is the lister output as good as or better than what the human came up with. Now, this will vary greatly depending on the sort of training sets the lister uses or the features settings, but as far as the comparer tool is concerned, for the most part the lister's AI did about as good a job as the human.

Experiment: Training Set compared to Same Script

This is a little bit more of a devious test. What happens if we feed in a single training set, and then use the exact corresponding script as the input? One would imagine that in an ideal world, the lister tool having the exact training set that goes with the raw movie, it would be able to line it perfectly. Although, in practice, this is not the case. This particular experiment gives us an idea of how good a job we might have done in picking our features. Better more refined features should inch us ever closer to ultimately getting a well-tuned set of features and settings.

Experiment: Feed Lister Output in as Input

This is a similar experiment as above, although, it should hit the nail even closer to the head. What happens if instead of feeding a training script into the vector populator tool, we actually feed in the output of the lister tool? This is possible since the output and training sets are

exactly the same format. The idea again is that if you feed in the raw script corresponding to the raw script you feed in to create that output, you should get results again that match almost exactly the same. And again, if the features are picked well, and tuned well, this would be the case.

Experiment: Many Training Sets vs Few Training Sets

In this experiment, we want to create several vectors, one which has a single script Training Set, one which uses several, maybe half the available Training Sets, and another which uses most if not all of the Training Sets. Then we want to feed the different vectors into the Lister tool with the same raw script each time. Then we compare the different outputs. The goal is to see if more training sets produces a better output or if we actually want fewer.

Experiment: Comparing Different Films by Same Director

This test is done to see how close the outputs are if the films involved are done by the same director. In order to complete this test, two different films by Quentin Tarantino were lined, Reservoir Dogs and Pulp Fiction. These are Tarantino's first and second movies and as such, have pretty similar styles. Here is exactly how the test works. We actually create two different vector files. One is based on a random set of the Training Sets that don't include one of Tarantino's films. The other vector is created with the first Tarantino Training Set. The lister tool is then fed the other Tarantino film in its raw script form along with each of the two vector files

in turn. We compare these two outputs against the Training Set of the second film and see which gets a better score.

Our hope is that the output that used the vector from the Tarantino Training Set will do better. This would indicate that the lister tool really can be influenced by the sort of Training Sets you feed it, and with a much larger number of Training Sets, one could be choosy enough to request a kind of output, and the lister would pick its own Training Set selection accordingly.

Round 1

Thirteen people ultimately participated in the Round 1 experiment. I created a table in an excel spread sheet, with columns representing the raters, and rows representing the experiments, and each cell representing the rater's top pick for each experiment. I pasted this data into the following website:

http://www.statstodo.com/CohenKappa_Pgm.php#Cohen%27s%20Kappa%20from%20rating%20scores

This gave me a Fleiss Kappa score of 0.2647 which according to

https://en.wikipedia.org/wiki/Fleiss'_kappa indicates "fair agreement".

I also tallied up the scores, and found that 111 out of 156 times, the human lined scripts were picked over the lister outputted scripts, for a percentage of 71.1538462%

Basic:

Unfortunately, the majority of the time, the human observers picked the sample from the training set (that is the shot list of the original movie) over the output of the lister tool. Although, a few people chose the lister output so it was not a complete failure. My theory on the original film preference is that having too many training sets in the creation of the vector file is what caused this outcome. Too many training sets from movies with different styles creates a sort of overly even affect. The shot list created is bland without too much variation. The humans picked up on this quickly and preferred the original film's output.

Control Sample:

The results of this experiment are slightly more encouraging. We really have two things going on here. One is, how well does the shot list of the actual movie fare against the shot list created by another person. And this was a very even split among the human judgers. The results do not say much about the shot lister tool, but it does reinforce the idea that good art is completely subjective. That is an important thing to keep in mind when reviewing the shot lister tool. The output may not at all resemble the original shot list of the actual movie, but as long as people see it as viable, that is what matters. With that in mind, the second thing is the output of the lister tool was not ranked last in enough cases to feel a certain amount of confidence about the output. Keep in mind, like the basic experiment, the output was created with a vector file containing all the training sets except the one for the movie tested, so, a more limited selection may have actually produced a better outcome.

Training Set Compared to Same script:

For the most part, the humans favored the training set over the shot lister output, but there were enough cases where the human favored the lister output to still have hope in the process.

Feed Lister Output in as Input:

This test actually had much more mixed results. It was in fact about half and half. The results of the training set may actually be viable enough as to use as a new training set themselves.

Many Training Sets vs. Few Training Sets:

These are actually some of the most encouraging results. Very few people actually picked the training set itself as their favorite. This could be that having four options made it more confusing, but on the other hand, it could also mean they all looked viable, and they decided to just go with any of them, also a good result. Also, on the whole, people seemed to prefer the outputs of the scripts that had less training sets going into the vector file.

Comparing Different Films by Same Director:

The training set was almost always preferred over the two lister outputs of this test, however, the lister produced by the vector with the same director was almost always preferred over the one created with mixed training set vector. This could be because the mixed one had more training sets which for the most part seem to water down the results, but it could also be that the one created by the same director training set actually does do a better job.

Round 2

I ran the experiment a second time after improving the algorithm by isolating each target feature and adding new features that pertained to shots. I tried as best as possible to replicate the first experiment with choices of scripts to use in the vector files and the output files. Unfortunately, only nine people participated the second time, however the results were encouraging.

This time, the Fleiss Kappa score was 0.1915 which indicates “slight agreement”. This is lower than the first round of “fair agreement” and could mean that it was more difficult for people to pick their favorite samples since they seemed to agree less. The tallied number of times a human lined script was picked was 72 out of 108, which is 66.66%, less than round 1’s percentage. Both this lower percentage and the lower Fleiss Kappa score suggest that the changes done to the algorithm between Round 1 and Round 2 of the experiments were an improvement.

Chapter: Conclusion

Although the main stated goal of this project was to create a program for generating shot lists from raw files, the portion of the project I truly found myself drawing a sense of satisfaction from was actually the Liner tool. I toiled over making the liner tool as good as possible, even to

the point of staying up many nights to improve it, well past the point that it was more than good enough to get the job done of creating Training Sets. It felt to me like a real tool I could actually use later on while actually making films. If I were to take anything away from this project, it would be to continue developing the liner tool into a professional grade app for use by film makers.

However, I do not mean to say all this as to diminish the final product of this project which is the lister tool. Although it may not truly measure up to artistic grade shot lists, it still produces both viable and worthwhile results. Throughout the experiments, the output was typically quite reasonable. Who is to say that an entire scene with six characters might not be one single static shot that is extreme close up? I have certainly seen movies do things just like that even stranger. Certainly, my refinement of splitting up the target feature calculation helped immensely. Although someone may not want to use this tool to create a final shot list, I do feel it creates something which someone could use as a starting place, and then use their own judgement to refine their shot list.

Which Training Sets are used as input really does make a difference too. In fact, using fewer Training Sets seems to have better results. Too many seems to sort of even out the melting pot so that no values are that unique creating a bland lined script output. And the set of features used also really makes a difference. Like using too many Training Sets, it appears that using too many features also watered down the results. This caused a sort of smoothing effect which tended to make every line the same.

In addition to further refining the features, adding new ones, with good weighting, the thing that would really improve this project would be something we knew from the beginning but did not have the time for and that would be having many more training sets. In the beginning, we even talked about thousands. With thousands of training sets, completely customized vectors could be created to suit many different styles of movie, genres. It could be filtered by director, actor, and cinematographer even. This sort of variation could really allow outputting shot lists that suit the human inputter's opinion of the movie, which at the end of the day, is really the most important thing.

References

Gatys, L. A., Ecker, A. S., & Bethge, M. (2015). *A Neural Algorithm of Artistic Style.* Tubingen, Germany.

Russell, S., & Norvig, P. (2010). *Artificial Intelligence A Modern Approach.* Pearson.