French 3730 Final Project

Tim O'Brien

Inspired by the work of William S. Burroughs, I decided to create a more modern version of his cut-up technique using Markov chain-based text generation in the C++ programming language. The cut-up technique involves cutting up pieces of a particular work or multiple works and reassembling them in a random order. This oftentimes leads to a result that isn't particularly interesting, because, while there may be a few unique or thought-provoking phrases, the large majority of the resulting text can be a big jumble of seemingly unrelated words. I wanted to consistently produce somewhat interesting results, i.e., that the entire result of my algorithm could be interpreted as its own standalone text that shares characteristics with the original text. Besides running very quickly, to the point of easily being able to use large novels as original texts with which to work from, I also wanted to be able to combine multiple authors' generated texts into one larger text that would share different characteristics of each author to see if any new meaning could be gleaned from looking at the particular authors as a whole versus individually. I ran the algorithm several times on different combinations of Beat authors and works to try to analyze both patterns in their works as well as ways to improve my algorithm.

The algorithm relies on Markov chain-based text generation. A Markov chain is a probabilistic model that relies solely on the last event to generate new events. There is no other history that goes into the model; it only cares about the present state which makes implementing text generation with it both effective and not as complicated as some of the other NLP (natural language processing) techniques—since NLP is a very active field of research and can get highly complicated by involving things like machine learning and artificial intelligence to study language patterns. By knowing the current state—and having previously analyzed the text—we can produce the next state (i.e. word) with no other information.

Analyzing the text is the first step in using the Markov chain text generation algorithm. We need to know the set of possible next states that can exist from our current state. In this

context, that means that for each sequence of words (prefix), let's say just 2 words to help illustrate the example, we want a list of possible follow-up words (suffixes) that ever appear after these 2 words at some point in the text. Another example will help illustrate—if my text is "The dog ate and the dog slept.", then the possible suffixes for "the dog" would be both ate and slept, because these 2 words appear directly after the prefix of "the dog". Something like "ate and" would just have one suffix in this case, the word "the", because there is only one occurrence of "ate and" and the word directly following it is "the". This is an uninteresting example, but we can see that if the text is longer or has some patterns in it, that we may have many possible suffixes for a given prefix. We can modify the prefix length if so desired, but generally either 2 or 3 is advisable; if we have a prefix length of 4 for example, there are many possible different prefixes that can appear in the text, so that one particular prefix will likely only appear one or two times. When we try to generate a text, it will largely follow the same format because there are not many suffixes to pick from, so, in all likelihood, we will end up picking the same suffix every single time and produce the same text every single time. Once the prefix length is established, we iterate through the text looking at each prefix and put the corresponding into a list of suffixes; two data structures in C++ help facilitate this sequence: a map and a vector. The map simply takes the prefix, "the dog" in our previous example, and creates a key for the prefix and a value for the suffix. Later, when we look up the particular key, the map will return the value for that key. The value ends up being a vector, which is just a fancy term for a list whose size can be increased if necessary. Whenever we find a particular prefix, we try to see if the key is already present in the map—if it isn't present, we create the key-value pair. We then add the particular suffix to the list of possible suffixes for that key (the prefix). Later, we can use the prefix to key into the map and randomly select one of the suffixes to use in our text generation. There are some other details that help make this initial processing and subsequent generation very fast and pain-free.

The first part of my program is parsing arguments passed to the program through the command line. I wanted to be able to make the program flexible, so that the size of the output file, the prefix size, and the actual authors from which to generate text could all be controlled through different arguments when the program is run from the command line. The size of the

output file is the only mandatory argument, the others default to a prefix size of 2 and simply using all the authors listed in the texts directory—this was done using a relatively new feature in C++17 (filesystem) that lets the programmer easily retrieve all the folder names in a given directory, so I just added each name to a list. I was somewhat familiar with parsing arguments from the last programming assignment I did in CS3251, so I pulled a lot of information from there. The program checks for the first argument, if there isn't one, it throws an error—since the size of the output file is a mandatory argument. It converts this argument to a number, because arguments to the command line in C++ are represented as C-Strings—effectively a bunch of letters—which can not be used in a numerical fashion without first performing some sort of conversion. Then, it goes case-by-case to see if the next arguments were either: not provided, only the prefix size was provided (which means all authors are used), only the list of authors was provided (which means that a prefix size of 2 is used), or the prefix size and a list of authors (each author separated by a space) was provided so that they can be correctly read in by the program.

The next thing that my algorithm does is that it takes these arguments and creates an options singleton with them. A singleton just means that I create an object—something that I can refer to by name and ask for certain information about through its methods of access—that can only be created once and can never be mutated after that fact. This is key when working with options, because we only want to actually create the options singleton the one time after we have parsed the different arguments passed to the program. The options should not change throughout the execution of the program; otherwise, we may end up with behavior that the user did not ask for if a new options object with different parameters is somehow created. A singleton both ensures that the options will not change—because the particular settings are only available to that specific part of the program—and also lets us easily access the options to use in future portions of the program with the aforementioned methods.

Creating the different source dictionaries for each author becomes incredibly easy when the options are properly handled. I store a list of all the authors inside my options object, which I then access to create a specific source dictionary for each author. The source dictionary is made using all the texts listed in the author's directory, i.e. for Ginsberg, I put "Howl" into the

directory first to generate the texts based on “Howl”—marked howl1, howl2, and howl3—then later added “Kaddish” to generate the text with Ginsberg and Corso. In the first instance, only “Howl” was used to create the source dictionary, in the second, both “Kaddish” and “Howl” were used. The algorithm opens the respective files and reads each, word-by-word, to create a mapping from a prefix to a list of suffixes as mentioned earlier. To make sure that the first and second words can be properly inserted into the map, an original prefix of (usually 2) n nonword characters was created, where n is the prefix size that the user specified earlier. In this instance, I used the | character as my nonword, since it is rare to find in texts. With a prefix size of 2, the map starts off by mapping “| |” to a list of suffixes, into which the first suffix—in our earlier example “The dog ate and the dog slept.” it would be “The” (capitalization and punctuation are included because they can alter the meaning of a word greatly)—is inserted. Then “| The” would be mapped to another list of possible suffixes, into which “dog” would be inserted and so on and so forth. A large optimization that I decided to make was to use an unordered map versus a standard map in C++. An unordered map is built using a hash table, which basically means that the particular key being used to access the map is passed through a mathematical function that generates a very large number that is used to identify where in memory the value is being stored. This allows us to perform very quick insertions and lookups on the map—invaluable because, in the case of novels like *On the Road,* we need to insert approximately 90,000 words. A normal map is built using a binary search tree; this object keeps track of the order of all mapped objects, which can be very useful, at the cost of some of the speed that the unordered map has. Since we do not care about the order that the prefixes are stored in, only their corresponding suffix, I chose to use an unordered map. The unordered map that is created from all the texts of the author is stored as a dictionary object that is passed by to the main portion of the program for further manipulation.

Once the program receives a list of dictionaries to process, it generates the output for each one and stores that in a list. Generating output is pretty simple, and in many ways can be viewed as the converse of the original algorithm to create the dictionary. It first looks for a starting word, in our examples, this has been marked by the “| |” prefix. This starting word then has a list of possible suffixes that it can choose from. One of those is randomly selected by

using a random distribution of integers that corresponds to the total number of suffixes for the particular prefix, so that each suffix has the same probability of being selected. In order to make sure that the selection was truly random, I created a random number generator using the current time as a seed—effectively a number from which to begin generating other random numbers using a built-in algorithm; however, since the time at each execution of the program is slightly different, the results can be drastically different because small changes in seeding can produce large changes in generated numbers. I then discarded the first 700000 entries of the random number generator because of a study published in 2006 that proved that even random number generators seeded with relatively small numbers—as this one is—can be made highly secure (and very close to truly random as opposed to pseudorandom) by discarding approximately the first 700000 generated numbers. Once the suffix was selected, it was written to the generated output for this particular dictionary. The suffix then kicks out the first word in the particular prefix and creates a new prefix for the next iteration of the algorithm, so if our prefix at the time was “dog ate” and our selected suffix was “and”, the following prefix would be “ate and”. This process is then repeated, with each selected suffix being written to the generated output and then replacing the first word in the prefix until the output reaches the desired length originally provided by the user. This output is passed back to the main function, where it is kept in the aforementioned list of outputs from each author.

Once we have our list of outputs from each author, there is not much left to do. So that the order of authors in the final text was not predetermined, I shuffled the list using the random number generator from earlier. After the shuffle completes, I write each author’s output to the physical text file in the output directory, and this produces the final result. This marks the end of the algorithm, so the program automatically terminates.

I used the algorithm to generate a few texts based on different combinations of works and authors. The first pairing that I looked at was just generating texts based on “Howl”. Since “Howl” is a very repetitive poem, I figured this would work very well because it would be able to very closely emulate the original structure of the poem. I chose to use a prefix size of 2, because using a prefix size of 3 would cause the poem to be largely the same as the original, since the source text is short. I ran the algorithm several times and picked 3 of my favorite

results. Each one highlights a different idea that is present in the main poem, but maybe not the main theme.

Howl 1 basically sounds like a homeless man ranting about the problems with America. As much as "Howl" sounds like that too, the rants themselves are fairly short before Ginsberg moves back into his standard "who yada-yada" structure. Those thoughts themselves are also fairly well-contained. The Howl 1 generated text just takes the main idea of disillusionment with society and turns the dial to 11. The structure is very staccato, much more so than "Howl" itself, and just highlights the relatively angry nature of the first part of the poem, which it mainly draws from. Since the 3 parts of "Howl" share much more in common with themselves than the other parts, the generated text has a tendency to stick to one part and this one chose to largely stick to the first part of the poem, almost parodying that part in short angry bursts.

Howl 2 is a really interesting generated text, and one that I was surprised to see. Most of the generated texts looked somewhat like Howl 1 or sometimes Howl 3 (or a slight variant with more emphasis on Carl rather than Ginsberg's mother). This one is almost totally different. It somehow focuses very thematically on homosexuality which is pretty interesting, since homosexuality is prevalent in "Howl" but more as a subtext to the main points of the poem. This one starts clearly "I saw the best minds of my cottage in the closet", and besides that cottage part it's implying that many of Ginsberg's peers were gay which we know to be true, but again isn't necessarily emphasized in the works or history of the time. It then goes on that the "soul is innocent and immortal" which seems to be a counterargument to the belief—prevalent in some Christian contexts mainly—that people can simply pick their sexuality i.e. choose to be gay; however, this is stating that the soul is innocent regardless of sexual behavior which is a very accepting message. Howl 2 finishes by describing the "heterosexual dollar"—that mainstream culture is based upon the idea of heterosexuality being the norm which is of course very true today—and then "the blond & naked angel" which is a reference to gay sex.

Howl 3 is probably my favorite text that I generated based on Howl. This theme was relatively common in the generated texts which is quite interesting because it doesn't seem to be as prevalent in the poem. Part 3 is of course about Carl Solomon, but Howl 3 decides to take

those same feelings towards Carl Solomon and instead direct them towards Ginsberg mother in a very "Kaddish"-like manner. It starts off with "I saw the best minds of my mother", which directly parodies "Kaddish" where Ginsberg is trying to remember her better times, when he was a kid. Then he flashes forward to Rockland "where your condition has become serious and is everywhere about us!". The rest of the poem talks about the general public perception of people like his mother and it ends with Ginsberg thinking back to Carl— "while you are not safe I am not safe"—and then presumably having sex with him ("imitate the shade of my cottage in the neck and shrieked with delight"). I don't necessarily think the themes that we see in Howl 3 are as present in the actual poem "Howl", but it's almost mind-blowing to see that maybe "Howl" and "Kaddish" aren't so far apart in their themes, because the generated text in this case just sounds so much like "Kaddish".

I then looked solely at texts based on Kerouac's *On the Road.* I don't think these two hold much meaning, but they're pretty interesting stories nonetheless, and it was cool to see how they turned out. I decided to use a prefix size of 3, because novels have a much more linear structure than poems so the longer prefix size serves it well. Furthermore, the generated dictionary is much, much larger, so there are many suffixes to draw from even with a prefix size of 3. Lastly, when I used a prefix size of 2, it just read like a bunch of garbage, so 3 seemed much more apt; 4 was also too similar to the original work and wasn't producing very interesting results. On the Road 1 talks about Kerouac going on a trip to the Congo and hearing the drums and the beat. It then actually says that this was all inside a restaurant, and so he and Dean go to sit down somewhere and take "28 Benzedrine". He then hallucinates about visiting Mexico City and potentially becoming a farmer or a cowboy, which is absolutely hilarious. On the Road 2 is very fragmented, first talking about Dean reading "her paper", while "she" watches him with "lowered and hating eyes", but then says that Dean reminds her of her late husband. After this presumed affair, Kerouac hitchhikes up some hill to some prairie where they watch cars go past. Most cars are filled with farmers, but a "hotrod" comes as well. Then it goes back to "she" listening to them followed by "she" and Dean strolling arm-in-arm. This "she" is never specified, but I'm going to assume it's the same person which makes the entire text really bizarre.

I finally tried combining two authors: Ginsberg and Corso; for Ginsberg, I used “Howl” and “Kaddish” as source texts and for Corso, “Mad Yak”, “Marriage”, and “Bomb”. This did not work so well. Both parts of the generated text don’t make much sense on their own, and they really don’t work well together. It basically goes from a passage about some man (unspecified) being crazy and slowly transitioning to how Ginsberg’s mother went crazy. Then it rapidly switches to literally one 150-word sentence about the bomb, and how Corso cannot hate it—citing the Greeks, Da Vinci, tomahawks, and German fairy tales. I’m not sure if Kaddish, and possibly all of Corso’s works, are just unsuited to producing interesting results with my algorithm, or if this maybe didn’t work well the few times I tried it. Regardless, I think it could be interesting with the right combination of authors. Kerouac and Twain could be very interesting for example—since *On the Road* parallels *Huckleberry Finn*—but I wanted to focus on Beat authors in this context.