Creative Coding - Week 5

Making Text Bots With Tracery

Lecturer: Dr Michael Cook (mike.cook@kcl.ac.uk)

Figure 1: A screenshot of a Bluesky bot which generates ideas for Bluesky bots.

This week we're going to focus on text, starting with one of the most common tools for making a text generator, *Tracery*, which we're going to use to make a bot that runs on the Bluesky social media service! Then, if there's time, I've got some extra tasks relating to processing text and making datasets for things like this.

1 Part 1: Generating Text With Tracery

Tracery is a library/framework for generating text using replacement grammars, designed by Dr Kate Compton when she was a PhD student (she is now Director of Play at LEGO and often visits London to see her team - you should apply for an internship with her!). Replacement grammars are a very common technique for text generation (and generating other things too) that have been used for many years, before the invention of computers. You can find the Tracery editor here:

http://tracery.io/editor/

The Tracery editor can be a bit weird sometimes. Tracery grammars are written in Javascript, but the editor itself is stricter than Javascript is. If it gets frustrating to use, I recommend downloading Tracery and using it locally.

You may have come across replacement grammars like the ones Tracery uses in your course if you studied topics like compilers. A replacement grammar is a series of rules like the following:

$$A \rightarrow B$$
$$B \rightarrow c$$

We use a grammar by starting with a symbol or sequence of symbols and then repeatedly replacing symbols using the rules in our grammar. If we started with the symbol A, we could replace it with

B using the first rule, and then replace *that* with c. A and B are what we call non-terminal symbols – the grammar doesn't stop working until it replaces all non-terminals.

Tracery grammars are written a bit differently, for clarity. Here's a rule from Tracery:

```
{
  "bird":["swan", "heron", "sparrow", "swallow", "wren", "robin"]
}
```

This rule says that the word 'bird' can be replaced with any of the words in the following list (Tracery will pick one at random). It's a bit like writing:

$$Bird \rightarrow swan$$
$$Bird \rightarrow heron$$
$$etc.$$

It's not exactly the same as this, but it's close. In our formal grammar example we used capital letters for non-terminal symbols. In Tracery, we use slightly different syntax. Look at the following:

```
{
  "start": ["What's that bird over there? I think it's a #bird#!"],
  "bird":["swan", "heron", "sparrow", "swallow", "wren", "robin"]
}
```

Notice that the first rule includes the word 'bird' twice. The first occurrence doesn't do anything - it comes out as 'bird'. The second occurrence is surrounded by #hash# symbols. This is a signal to Tracery to go and look for a rule that has this pattern at the beginning. If we run this grammar on the starting sequence '#start#' we get something like:

```
"What's that bird over there? I think it's a wren!"
```

Traditionally, we call our starting pattern 'origin' instead of 'start'.

1.1 Tracery Features

This is all you need in a replacement grammar, but Tracery adds a few extra features on top.

1.1.1 Capitalisation

We can add `.capitalize` after a pattern name to capitalise what comes out of it. This is useful when patterns are used both at the start of sentences and the middle. For example:

```
{
  "origin": ["#fruit.capitalize#? Oh, I love #fruit#!"],
  "fruit":["bananas", "oranges", "apples"]
}
```

This will capitalize the first use of the `fruit` pattern, but not the second one. Without this, it would print something like:

```
bananas? Oh, I love bananas!
```

Which looks a bit uglier (and is more obviously generated by an algorithm).

1.1.2 Persistence

If you run the above example you'll notice that the two different references to the `fruit` pattern return different fruits, but ideally we want it to return the same ones. We can get around this by assigning the output of a pattern to a variable.

```
{
  "origin":["[f:#fruit#] #f.capitalize#? Oh, I love #f#!"],
  "fruit":["bananas", "oranges", "apples"]
}
```

The first part of the first rule, `[f:#fruit#]`, doesn't produce any output. Instead, it calls the `fruit` pattern, binds it to the variable `f`, and that means we can use it twice in that pattern and get the same output. We can now reference the same thing multiple times. This works anywhere in the grammar too - other rules can refer to `f` as well.

1.1.3 Contextual Formatting

Suppose we wanted to refer to our fruit using the indefinite article:

```
{
  "origin":["Man I would love a #fruit_singular# right now."],
  "fruit_singular":["banana", "orange", "apple"]
}
```

This is grammatically incorrect if we choose a fruit that begins with a vowel ("a apple"). Tracery has a few rules for fixing common patterns like this. Here are some of them:

```
{
  //Okay, now our fruit pattern is singular words
  "fruit":["banana", "orange", "apple"],

  //Adding '.a' automatically inserts 'a' or 'an' as appropriate.
  "article_example":["Man I would love #fruit.a# right now."],

  //Adding '.s' will attempt to automatically pluralise the noun
  "plural_example":["Man I would love some #fruit.s# right now."],

  //Adding '.ed' will attempt to conjugate a verb
  "verb_example":["Maybe some #prepare.ed# #fruit# too."],
  "prepare":["peel", "chop", "freeze"]
}
```

Note that these helper patterns aren't perfect – for example, it'll turn 'chop' into 'chopped', but 'freeze' will become 'freezed' rather than 'frozen'. But they can still really help write patterns quicker.

1.1.4 Weighting Tricks

Tracery selects from a pattern uniformly, meaning every pattern has an equal chance of being picked. If we want to mess with this, we have a few options:

```
{
  //We can change probabilities by copying and pasting words to make them more
    likely to appear (and other words less likely)
  "origin":["I had #fruit.a# for breakfast today."],

  "fruit":["orange", "orange", "orange", "apple", "apple", "apple", "pineapple", "
    mango"]
}


{
  "origin":["I had #fruit.a# for breakfast today."],

  //We can also add sub-patterns. This has a bunch of common fruits and a
    subpattern
  "fruit":["orange", "apple", "banana", "grapes", "#rarefruit#"],
  //Each of these fruits are half as likely to appear as the ones in the main
    pattern
  "rarefruit":["pineapple", "mango"]
}
```

1.2 Challenge: Sketch A Draft Text Generator

In this session we're going to be building a text generator and then turning it into a bot that posts on the Bluesky social media site. For the first part of this session, I'd like you to think of an idea for a simple text generator, and write a first draft. Try to think of a bot that has at least **4-5 Tracery rules**.

If you'd like some inspiration, you can find some examples of my favourite Tracery bots in Figures 2 and 3. In the spirit of today's session, I've also written a bot that generates ideas for Tracery text generators. You can find it here, maybe you'll see an idea you like:

@botideabot.bsky.social

2 Challenge: Bots Done Quick

In the next section we'll look at using text processing tools to help augment or create text bots. For now, let's focus on getting your bot online! Bots that are pure Tracery can be set up very quickly through a service called Blue Bots, Done Quick! Find it here:

Blue Bots, Done Quick!

The website has very clear instructions on how to set up your bot, but I'm going to go through it here too in case it's easier for you.

First, head to https://bsky.app/ and sign up for an account for your bot. If you already have a bsky account and are logged in, go to Settings → Add Account. When giving your email address, if

(a) Foley Artists has a two-part pattern: an example of a material you could make a noise with, and then a sound it is mimicking. The two parts are unrelated, which creates fun contrasts. Both lists use evocative and unusual language and examples.

(b) Magic Realism Bot creates one-sentence premises for stories that have supernatural, mystical or magical elements contrasted with everyday life. This bot has a lot of patterns and subpatterns which keeps it fresh.

(c) This is based on a generator written by computer scientist Gordon Strachey in the early 1950s. More here.

Figure 2: Some example Twitterbots, all built using Tracery.

(a) You can also make Tracery grammars with things other than words - like Unicode symbols or emoji. This one builds little scenes of people in galleries.

(b) For super-advanced stuff, Tracery can actually create SVG files (vector graphics files which describe images using XML). This bot created landscape scenes using a Tracery grammar.

Figure 3: Some example Twitterbots, all built using Tracery.

it's a Gmail account you can add a suffix to your email to help filter emails sent to the bot account. For example, if your email is `mike@gmail.com` you can give your email as `mike+bot@gmail.com` and it will still come through to you, but it makes it easier to filter.

When naming your bot, it's customary to include the word 'bot' in the title, but not compulsory. However, it is good practice to declare that your bot is a bot in its description, and also put your name or some way of contacting you in case anyone has any problems with how your bot is behaving.

Once you've created your bot, go to Settings → Privacy and Security → App Passwords, and create a new password. It doesn't matter what you call it. **Make a permanent note of this password somewhere.** You can never view this password again, so if you lose it you'll need to regenerate it and reconfigure the bot. I recommend emailing it to yourself. After you're done, copy the password - you'll need it for the next step.

Head to Blue Bots, Done Quick! and click Create A Bluesky Bot. Type in the full name of your bot (it will be something like 'mybot.bsky.social'. On the next page, paste in the app password you just created.

Figure 4: The bot editing interface for Blue Bots, Done Quick!

You'll now see the bot editing page. If you click the 'JSON View' tab, you'll be able to paste in your Tracery grammar. This page lets you see test outputs from your bot, define how often it posts, and change other settings. When you're done, hit the green button at the bottom of the page – congratulations, your bot is now online!

3 Extension: NLTK

For this part of the session you'll need a Python package called NLTK - the Natural Language Tool Kit. The easiest way to install it is to type the following into a terminal:

```
pip install nltk
```

If you have any problems installing NLTK, just let me know. NLTK is a very full-featured project that has been in development for many, many years and contains all sorts of useful tools for working with text data. It's my first port of call when I'm doing anything with text. We also need to install an extra thing as part of NLTK - something called `cmudict`. This is a dictionary dataset created by Carnegie Mellon University researchers (hence the name) which we'll be using today. To do this, in a terminal type `python` to open the Python prompt, then run the following two lines one after the other:

```
import nltk
nltk.download('cmudict')
```

NLTK should download and install the dictionary. You can type `exit()` to exit Python after this is done. You may be asked to install other modules by NLTK as we go, you can use the same approach. From now on, we'll write our Python code in files and run them from the terminal.

3.1 Processing Text with NLTK

NLTK can do a load of cool things, but I wanted to show you a few examples today to help give you some inspiration for future text projects. If you want to make a more complex bot that you run off your own server, you can have it do stuff with NLTK when it posts, for example.

For the examples below, I've included two small datasets you can use as input which you can find with the supplementary files on the same site you got this PDF from. The first, pop-movies, is a processed and cleaned up version of one I found on Corpora. Corpora is a GitHub repository run by Darius Kazemi, who was a prominent member of the Twitterbot community back in the 2010s. People contributed interesting data to the repository for others to use for their botmaking. Here's a sample of the movie list:

```
Gone with the Wind
The Wizard of Oz
One Flew Over the Cuckoo's Nest
Lawrence of Arabia
```

The second dataset is a small set of six reviews from IMDB's open-source dataset of reviews (the full set is on Kaggle and HuggingFace and has 50,000 reviews in). I've just taken a few to show you some of NLTK's features.

If you've not read data in from a file in python before, I do it in quite a messy way so it's not best practice, but you can find a little skeleton template in a file called process.py. Here's a snippet:

```
reviewlist = open("movie-reviews.txt").readlines()

for review in reviewlist:
    #do something here
```

3.1.1 Part-of-Speech Tagging

So what can we do with this? Well we can ask NLTK to do something called *part-of-speech tagging* where it labels different words according to what kind of word they are (noun, verb, etc). Let's try this on a movie review:

```
reviewlist = open("movie-reviews.txt").readlines()

```

```
# breaks the string into separate word token
words_as_tokens = tokenize.word_tokenize(movies[0])
# tags each token as a part of speed
tagged_tokens = nltk.pos_tag(words_as_tokens)
# we can then print some of them out
print(tagged_tokens[:10])
```

This gets us an output like this:

```
[('A', 'DT'), ('wonderful', 'JJ'), ('little', 'JJ'), ('production', 'NN'), ('.', '
    .'), ('The', 'DT'), ('filming', 'NN'), ('technique', 'NN'), ('is', 'VBZ'), ('
    very', 'RB')]
```

Each token is a pair of strings - the first is the token, like 'wonderful', and the second is the POS tag, like 'JJ' which is an adjective. You can find a list of POS tags online here.

This means we can do things like filter out all the nouns from a movie review:

```
['production', 'filming', 'technique', 'fashion', 'comforting', 'sense', 'realism'
    , 'piece', 'actors', 'Michael', 'Sheen', 'polari', 'voices', 'seamless', '
    references', 'Williams', 'entries', 'watching', 'piece', 'production', 'master'
    , 'comedy', 'life', 'realism', 'home', 'things', 'fantasy', 'guard', "'dream",
    'techniques', 'knowledge', 'senses', 'scenes', 'Orton', 'Halliwell', 'sets', '
    Halliwell', 'murals', 'surface']
```

As you can see, this isn't always perfect - 'seamless' is not a noun, it's an adjective.

3.1.2 Sentiment Analysis

Sentiment analysis means trying to assess the emotional weight and strength of a piece of text - is it positive or negative, happy or sad, and how intensely? There are a lot of ways this has been attempted over the years, and it became a popular research area partly because it was very valuable for businesses (advertising, customer service, and so on). NLTK has some fancier sentiment analyses built in but we can also use a nice and simple one.

```
from nltk.sentiment.vader import SentimentIntensityAnalyzer

#read reviews in
reviews = open("movie-reviews.txt").readlines()

#create the analyser
sia = SentimentIntensityAnalyzer()
#score the review
ss = sia.polarity_scores(reviews[0])

#print the results
print(reviews[0])
for k in sorted(ss):
  print('{0}: {1}, '.format(k, ss[k]), end='')
```

What we get from this is a set of four scores. Three of them are for individual qualities (positive, negative and neutral), with a fourth 'compound' score that you can think of as an overall weight, from -1 to 1.

```
compound: 0.9641, neg: 0.055, neu: 0.768, pos: 0.177
```

Sentiment analysis is flawed. It should not be relied upon to read human emotion in important situations, especially where there are legal, ethical, moral or other concerns! But it's great for

creative coding, art projects and other explorations of technology. As with all technology, take the answers it gives you with a pinch of salt.

3.1.3 Rhyme Detection

A task I frequently needed to do when I was making silly text stuff was a rhyming dictionary. The way I used to do this was to scrape an online rhyming dictionary every time I wanted to find a rhyme, but NLTK can help us make it easy! NLTK can load in something called the CMU Pronounciation Dictionary (it's also present in other python libraries like `pronouncing` if you prefer that).

In `movies-swap.py` I've implemented a function called `getRhymes` that returns a list of rhyming words for a given target word. It takes a number too which is how strong the rhyme is – 1 will just target the last syllable, 2 the last two syllables and so on.

3.1.4 Putting It Together: Puns

I got really obsessed with pun generation during my PhD. I made pun generators for Twitterbots, for my AI games research, and more. We now have a few tools to make simple puns! We just need one more dataset - in this case, `animals.txt` which is a list of animals. Let's change some movies so they're about animals instead.

Here's the step-by-step of what we're going to do:

- Pick a movie.
- POS-tag it to find a noun.
- Find a set of rhymes for the noun.
- Check to see if any of our animal names are in the list.
- If they are, replace the words in the movie with the rhyming animal word.

You can find the code for this in `movies-swap.py`. Let's run through a quick example:

```
The Wizard of Oz
```

If we get NLTK to tag this, we get two nouns: `Wizard` and `Oz`. Let's take Wizard first and get a set of rhymes:

```
{'izzard', 'blizard', 'hazzard', 'biohazard', 'izard', 'buzzard', 'hazard', '
    wizard', 'blizzard', 'lizard', 'haphazard', 'gizzard', 'grizzard', 'quizard', '
    edzard'}
```

Then we go through our set of animals and check to see if any are in this list. We find one that is: `lizard`. Swapping this back into the movie, replacing the original noun, we get:

```
The Lizard of Oz
```

Not bad, right? Not all of them are this good. My method of searching the rhyme lists and so on is very inefficient also, this was hacked together very quickly (as all good creative code should be).

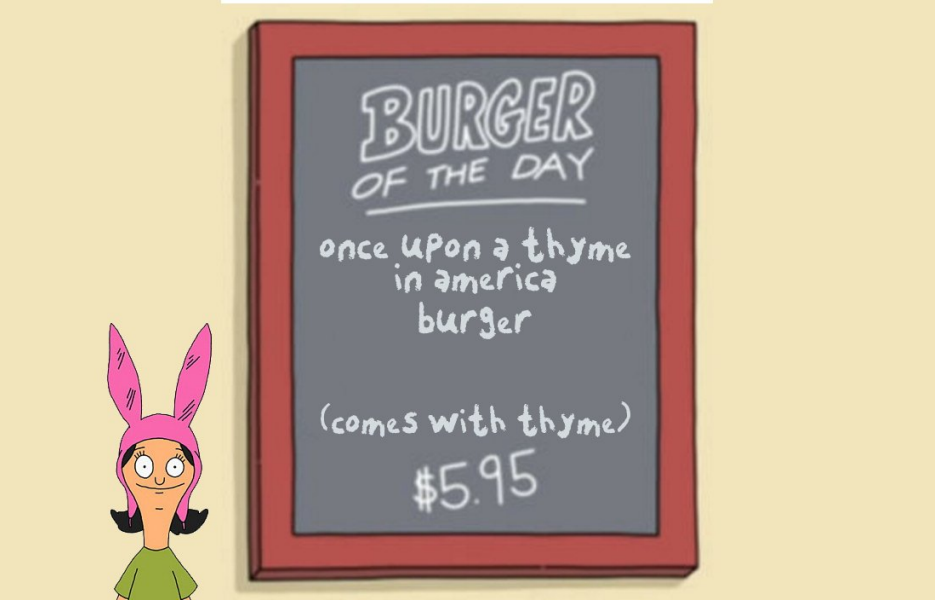

Figure 5: A tweet from the Bot's Burgers Twitterbot. This is a pun on 'Once Upon A Time In America'.

Hopefully this has illustrated how easy it is to do really fun stuff with just a few simple datasets and some python libraries! I've included a few extra datasets here too, taken from Corpora.

This approach inspired my personal favourite Twitterbot I made, Bot's Burgers. In the TV show Bob's Burgers, the protagonist always has a burger of the day that has a pun name related to the ingredient and some kind of pop culture reference. I built a dataset of famous books, films, videogames and proverbs, combined it with a set of ingredients and custom phrases, and then had a Twitterbot assemble this together with some custom art and fonts to create generated burgers. All using a very similar approach to the one I outlined here!

4 What Next

4.1 Share your work with me!

I'd love to see what you make, and I'd also love to be able to share people's work at future department events, or show to students in future years for inspiration! I would love it if you shared what you make - anonymously if you prefer, or with your name for full credit.

Ways you can share your work:

- Email the Tracery grammar, or a link to your bluesky bot, to me directly (mike.cook@kcl.ac.uk)
- Post it to GitHub and share with me (username: gamesbyangelina)
- Drop it anonymously in this form: https://forms.gle/qh8UWnqW1fNJyfxz8.

4.2 Further Reading

A lot has been written and spoken about Tracery and about text generation. Blue Bots Done Quick is a successor to Cheap Bots Done Quick, which ran over 50,000 community-made bots on Twitter (sadly killed off by API changes).

If you're interested in the design of Tracery, its creator Kate Compton wrote her PhD thesis about the design of what she calls 'casual creators' - creative technology tools that support a specific kind of fun, playful and low-stakes creativity. It's a good thesis, you can read it here.

If you want to get into writing generative text in this way, the absolute best to ever do it, in my opinion, is Emily Short. Short is a narrative design, game engineer and writer who has done a lot of amazing things across the games industry. The work I want you to check out is Annals of the Parrigues, which she made for a novel-generating competition. The aim is to generate a document of at least 50,000 words that makes sense as a whole thing. Her entry was a travel guide for a place that doesn't exist. The guide itself is really rich and interesting, but the thing I really recommend you read is the appendix. It's only a dozen pages or so, but it outlines her thinking about generating text and how to be creative through that. It's one of the best things anyone has ever written about generative text.