UCD / School of Engineering and Architecture / Electrical and Electronic Engineering /

Assignment One

Applications of text analysis

This report will:

- With songs from two different artists (King Crimson and Johnny Cash) use the tools introduced in the module to identify similarities and differences in the style of language used.
- Identify defining features of the above artists and assess their utility in guessing an unseen song.
- Discuss a context where this analytical approach could apply to another task related to the author's own academic background, electronic engineering.
- Discuss necessary additions or alterations to the approach to better perform the above-mentioned task.

COMP40020 Human Language Technologies

Authors : Tiarnach Ó Riada 16315466
Date submitted : 10th March 2020

Contents

List of Figures

List of Listings

1 A Comparison of Tendencies

I attempted to distinguish between the linguistic tendencies of King Crimson and Johnny Cash—progressive rock and folk/country artists respectively—using the tools learnt in this module.

1.1 Gathering and processing of data

1.1.1 Retrieval of lyrics

1.1.1.1 Download The songs were downloaded from genius.com with the code shown in listing 10. The api used is documented in [@br20]. The structure of the function shown in listing 10 is as follows:

`ll:2-8` Setup api metadata and song storage location.

`ll:10-15` The song list for each artist was set up to be accessed page-by page, with the maximum page size being fifty songs. Here, the first page is retrieved, using the python `requests` library.

`l:16` The data structure returned by the `GET` request for an artists songs has a field corresponding to the index of the next page, if one exists. Here, that index is used to test whether or not the last page has been processed.

`ll:17-19` Sim. ll. 13-15.

`l:20` Advance the page count for the next iteration.

`ll:22-32` A loop over those songs returned that match the id of the artist requested and whose lyrics were complete, i.e. not in progress. If the song has not already been downloaded—no file exists that matches its proposed filename—it will be downloaded as shown, using the `BeautifulSoup` library [bea20] to retrieve the lyrics from the webpage at the url retrieved from the earlier json object.

1.1.1.2 Presentation as useful data I used a pair of classes, `Song` and `Artist` (listings 2 and 11), to perform the filtering and organisation of the lyric data.

The `Artist` class (listing 11) essentially just creates a list of song objects from filenames that match a certain pattern corresponding to one of its songs.

The `Song` class (listing 2) has more functions, some of which are used later for the analysis of the artists' work.

`2-8` `__init__`
Constructor of `Song`. Stores pre-downloaded lyrics as tokenised words. The weakness of this approach is that `nltk.word_tokenise` splits on punctuation, such that words like "can't" and "don't" become `['can',"'t"]` and `['don',"'t"]`.

`10-14` `from_file`
Alternate factory method of `Song` to construct the class from a given filename, parsing the title and artist from said filename.

`16-17` `filtered`
Render the raw token list of the song's lyrics into a form more suitable for analysis:

- Filter out punctuation. The analyses performed operate on words; the idiosyncrasies of each artists' punctuation styles are irrelevant. Were analyses to be performed on, for example, how the artist might begin a sentence, this step would be removed.
- Filter out words that do not add information but rather facilitate communication between people. These words would not help to spot any meaningful distinction between the artists' styles or help find some theme in their work as they are common to most works.

`22-25` `filename_fmt`
Return a path that will match a file or files corresponding to the author and title given.

Once downloaded, a list of artists was compiled from the downloaded lyrics as shown in listing 1.

```
1 artists = ["King Crimson","Johnny
  ↪  Cash"]
2 aa = [Artist(a) for a in artists]
```

Listing 1: Creating a list of artists. For the definition of `Artist` see listing 11.

1.2 Analysis

Once the artists' data had been rendered into a usable format, the analyses could be performed.

Some analyses were performed on the entirety of an artist's recorded body of work. These corpora were generated as shown in listing 3.

1.2.1 Methods

While some of the below methods of analyses provided good insight into the distinguishing features of each artist's work, combinations proved to be the most helpful, for example stemming words before applying a frequency distribution to the stemmed words.

1.2.1.1 Frequency Distribution The `nltk.FreqDist` class constructs a frequency distribution of a list of tokens, i.e. a set of words and their corresponding number of occurrences in the list of tokens. Its plot method was often used in this assignment. Plots of the frequency distribution of words from each artist's work are given in figs. 1a and 1b.

1.2.1.2 Stemming is the process of finding a root form or stem for each word in order to reduce or ideally eliminate duplicates from consideration. Listing 4

```python
class Song(object):
    def __init__(self, title, artist):
        self._title = title
        self._artist = artist
        path = self.filename_fmt(self._artist, self._title)
        print(f"SONG: {self._title} / {self._artist} / {path}")
        self._file = open(path, "r")
        self.words = nltk.word_tokenize(self._file.read())

    @classmethod
    def from_file(cls, filename):
        m = re.match(r"([^_]+)(?:_)([^_]+)(?:\.song)", filename)
        g = m.groups()
        return cls(g[1], g[0])

    def filtered(self):
        return [t.lower() for t in self.words if t.isalnum() and (t.lower() not
        ↪  in stop_words) and (t not in punctuation)]

    def freq_dist(self):
        return nltk.FreqDist(self.filtered())

    @staticmethod
    def filename_fmt(artist, title = None,):
        titlestr = str(title) if title is not None else "*"
        return Path(f"{artist}_{titlestr}.song")
```

Listing 2: Listing for `Song`, the class managing retrieval and processing of song lyrics.

```python
artist_bow = [None] * len(aa)
for i,a in enumerate(aa):
  words = []
  for s in a.songs:
    words.extend(s.filtered())
    artist_bow[i] = words
```

Listing 3: Generation of master list of artists' lyrics.

shows the procedure followed throughout this assignment. Below, where a variable has been suffixed with `_stem`, the words in that list have undergone this process.

```python
porter = nltk.PorterStemmer()
artist_bow_stem =
  [[porter.stem(w) for w in bow]
   for bow in artist_bow]
for stem in artist_bow_stem:
  nltk.FreqDist(stem).plot(15)
```

Listing 4: Stemming and its effect on the frequency distribution of the artists' lyrics. Figures 1c and 1d show the new frequency distributions after stemming.

1.2.1.3 Lexical Diversity Lexical diversity, as defined in listing 5, is the ratio of the number of unique words in a text to the length of said text. It is a measure of the complexity of the text.

I aimed to use it as a means of distinguishing songs from the different artists on the premise that one artist's work would be more lexically diverse than the other. This premise seems to have proved true: although I haven't assessed the degree of the distinction, the lexical diversity of King Crimson's work—when taken over their whole corpus—is 68.53% greater than that of Johnny Cash's (see listing 6).

A similar relationship exists when lexical diversity is evaluated over each song (listing 7). Two measures of central tendency, the arithmetic mean and the median, show a significantly greater average lexical diversity for King Crimson's work.

If the lexical diversity of the songs is normally distributed, then due to the lower mean and smaller standard deviation, the work of Johnny Cash is once more less lexically diverse than that of King Crimson; these normal distributions could be used to predict the likelihood of a given song belonging to one or another artist.

One factor that reduces the predictive power of lexical diversity is that I didn't filter the songs to exclude those that were instrumental—these had a lexical diversity of one as the site denoted their lyrics as the string `[Instrumental]`—and therefore boost both mean and median values. Only King Crimson have instrumental songs, and of these approximately twelve.

```python
def lexical_diversity(text):
  return len(set(text)) / len(text)
```

Listing 5: Definition of lexical diversity.

```python
for art,bow in zip(artists,
↪  artist_bow):
print(f"LEXICAL DIVERSITY OF {art}:
↪  {lexical_diversity(bow)}")
```

Artist	Lexical Diversity
King Crimson	0.2819
Johnny Cash	0.0887

Listing 6: Overall lexical diversity of artists.

1.2.1.4 Collocations Listing 8 shows the code used to find collocations in both artists' bodies of work and listing 9 shows the collocations found[1]. These proved to be a great means to distinguish the two artists: among the words most correlated in King Crimson's oeuvre being the more violent bigrams and trigrams of `('greed','poets')` and `('barbed','wire','politicians')`; the collocations resulting from Johnny Cash's work on the other hand had a more folk, religious and American country music feel with examples like `('redeemer','beckoning')` and `('waving','patriotic','nephew')`.

This means of measurement seemed to provide the most distinction between the artists. Given the most correlated bigrams in a new song, I feel I would be able to guess reasonably accurately to which artist it belonged.

```python
import statistics as stats
for a,llex in zip(aa,lexdiv_stem):
  print(f"==>ARTIST:{a.name}")
  print(f">MEAN {stats.mean(llex)}")
  print(f">SD {stats.stdev(llex)}")
  print(f">MEDIAN{stats.median(llex)}")
```

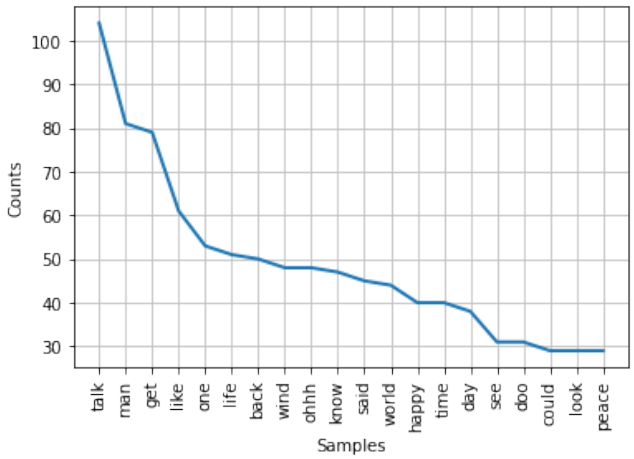
Measure	King Crimson	Johnny Cash
$\bar{x}$	0.7250	0.6580
σ_x	0.2116	0.1428
$median\{x\}$	0.7554	0.6724

Listing 7: Statistical lexical diversity over individual songs using the `statistics` library (top); output statistics (bottom).

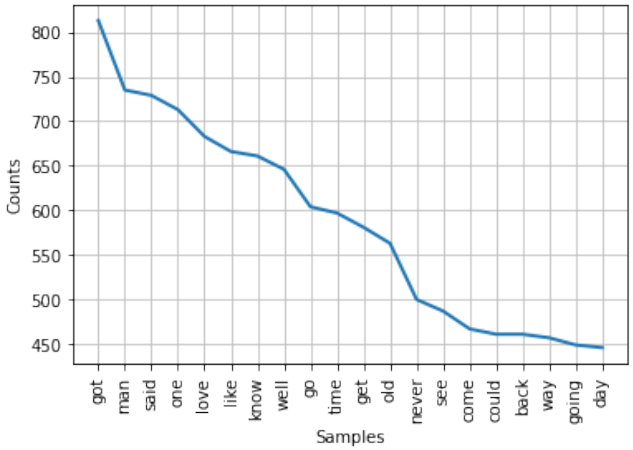
```python
bigram =
↪  nltk.collocations.BigramAssocMeasures()
trigram =
↪  nltk.collocations.TrigramAssocMeasures()

bfinders =
↪  [BigramCollocationFinder.from_words(bow)
↪  for bow in artist_bow]
tfinders =
↪  [TrigramCollocationFinder.from_words(bow)
↪  for bow in artist_bow]
for finder in bfinders:
  finder.apply_freq_filter(5)
  print(finder.nbest(bigram.pmi, 10))
for finder in tfinders:
  finder.apply_freq_filter(5)
  print(finder.nbest(trigram.pmi, 10))
```

Listing 8: Finding collocations in both bodies of work. For output, see listing 9.

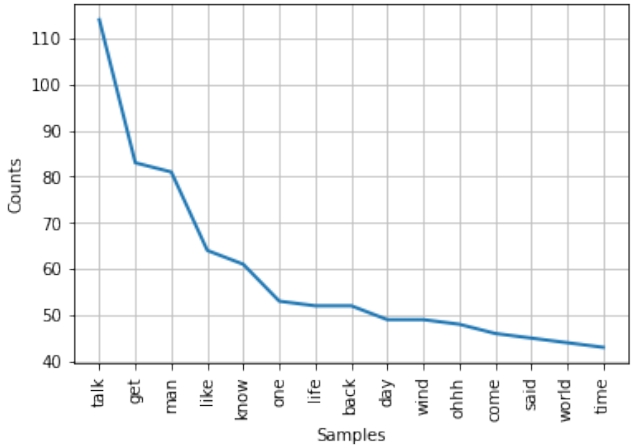
[1]This code is adapted from some at [col20]. The idea to use collocations comes from Workshop One.



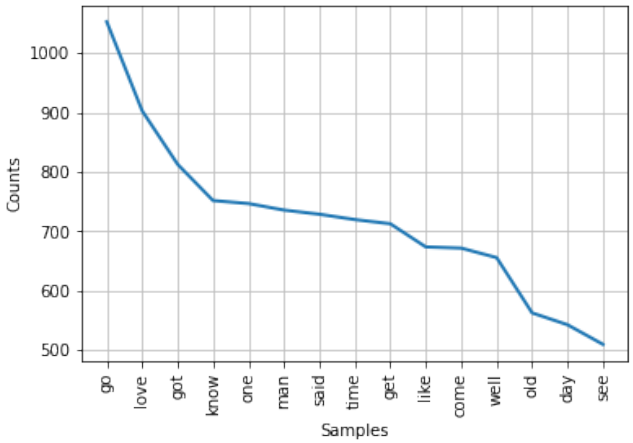
(a) King Crimson, filtered



(b) Johnny Cash, filtered



(c) King Crimson, filtered, stemmed



(d) Johnny Cash, filtered, stemmed

Figure 1: Frequency distributions of words in artists' bodies of work with various methods of preprocessing applied.

```
# King Crimson
[('greed', 'poets'), ('innocents', 'raped'),
('poets', 'starving'), ('pyre', 'innocents'),
('schizophrenic', 'tendencies'), ('barbed', 'wire'),
('paranoia', 'poison'), ('raped', 'napalm'),
('wire', 'politicians'), ('prince', 'rupert')]
#Johnny Cash
[('boa', 'constrictor'), ('chandeliers', 'burgundy'),
('redeemer', 'beckoning'), ('sandy', 'scag'),
('patriotic', 'nephew'), ('honky', 'tonk'),
('rick', 'rubin'), ('orange', 'blossom'),
('wondrous', 'waddy'), ('wooly', 'booger')]
# King Crimson
[('greed', 'poets', 'starving'),
('pyre', 'innocents', 'raped'),
('barbed', 'wire', 'politicians'),
('innocents', 'raped', 'napalm'),
('many', 'schizophrenic', 'tendencies'),
('funeral', 'pyre', 'innocents'),
('politicians', 'funeral', 'pyre'),
('poets', 'starving', 'children'),
('scream', 'paranoia', 'poison'),
('wire', 'politicians', 'funeral')]
# Johnny Cash
[('crystal', 'chandeliers', 'burgundy'),
('produced', 'rick', 'rubin'),
('waving', 'patriotic', 'nephew'),
('patriotic', 'nephew', 'uncle'),
('orange', 'blossom', 'special'),
('drunken', 'ira', 'hayes'),
('drifter', 'driftin', 'drifter'),
('slapped', 'shoe', 'polish'),
('ends', 'tie', 'binds'),
('flag', 'waving', 'patriotic')]
```

Listing 9: Collocations found in bodies of work. Output of listing 8.

```
def get_songs(artistid, song_dir = "songs"):
  song_dir = Path(song_dir)
  url = f"https://genius.p.rapidapi.com/artists/{artistid}/songs"

  headers = {
    'x-rapidapi-host': "genius.p.rapidapi.com",
    'x-rapidapi-key': "8c62b01aa3msh79ca17d82d4e5dep1a3ddejsn9346cf9aa3cb"
  }

  stepsize = 50
  page = 1

  response = requests.request("GET", url, headers=headers,
  ↪  params={"sort":"title","per_page":str(stepsize),"page":str(page)})
  song_page = response.json()['response']['songs']
  songs = song_page
  while response.json()['response']['next_page']:
    response = requests.request("GET", url, headers=headers,
    ↪  params={"sort":"title","per_page":str(stepsize),"page":str(page)})
    song_page = response.json()['response']['songs']
    songs.extend(song_page)
    page = response.json()['response']['next_page']

    for s in songs:
      if s['primary_artist']['id']==artistid and s['lyrics_state']=='complete':
        path = song_dir.joinpath(
        ↪  f"{s['primary_artist']['name']}_{s['title'].replace('/','')}.song")
        if not path.is_file():
          soup = BeautifulSoup(requests.get(s['url']).content, 'html.parser')
          lyrics_tag = soup.find('div', class_='lyrics')
          if lyrics_tag is not None:
            lyrics = lyrics_tag.get_text()
            f = open(path, "w")
            f.write(lyrics)
            f.close()
return songs
```

Listing 10: Listing for `get_songs`, the function to retrieve and store songs.

```
class Artist(object):
    def __init__(self, name, song_dir = 'songs'):
        self.name = name
        p = Path(song_dir)
        self.songs = [Song.from_file(str(n)) for n in
        ↪  p.glob(str(Song.filename_fmt(self.name)))]
```

Listing 11: Listing for `Artist`, the class managing the song list belonging to each artist.

2 Application

My academic background is that of electronic engineering.
Digital hearing aids can allow the user to program the sounds they hear [dha87]. The basis for their operation is that they can, thanks to digital signal processing, selectively amplify various frequencies. By extension, they can react to combinations of these frequencies that form words.

2.1 Proposal

I propose that an application of the analyses performed in this assignment is the automatic filtering of advertising audio at the input channel. This could be performed in such a manner that the user could still hear useful signals such as conversations relatively well, while the noise of the advertisement is significantly attenuated.
This could be done by a combination of the above analyses, for example:

2.1.1 Bigrams

After analysing the input words for some time, if there is a high count of bigrams that correspond to advertising content such as `('low','prices')` or `('amazing','deals')` perhaps, that audio source will be flagged for attenuation.

2.1.2 Frequency distribution

Advertising may have a characteristic frequency distribution of words. If such a frequency distribution can be identified, perhaps learned with some form of machine learning algorithm, then word sources such as a tv playing an ad can be identified—by the same learned model—and filtered out.

2.2 Necessary Additions to Previous Analyses

For a constrained-time application such as live ad-blocking, the approaches studied above are less than ideal. This is due to their needing a large input sample to filter a noise source with any degree of confidence. Better approaches would involve perhaps a machine-learning model that makes quicker predictions based on less data; such a model would have to be extensively trained on a wide variety of training data in order that the hearing aid continue operating normally for most non-intrusive signals.

References

[bea20] Beautiful Soup Documentation, March 2020. URL: `https://www.crummy.com/software/BeautifulSoup/bs4/doc/`. 1.1.1.1

[@br20] @brianiswu. Genius API Documentation, March 2020. URL: `https://rapidapi.com/brianiswu/api/genius`. 1.1.1.1

[col20] NLTK Collocation Tutorial, March 2020. URL: `http://www.nltk.org/howto/collocations.html`. 1

[dha87] Sep 1987. URL: `https://www.rehab.research.va.gov/jour/87/24/4/pdf/graupe.pdf`.