Introduction

This is a worksheet for the fifth `COMP3071` lab.

Language is a devious politician; it does not always say what it means. Figurative language—metaphor, simile, exaggeration, sarcasm, analogy, etc.—are an important part of how language works. The behaviour of a language agent can be enhanced by using external sources of information about language. In this practical session, we will explore how a web-based service that provides metaphorical language can help us add figurative language to our text. Along the way, we will get into details about text handling and natural language processing in Python, which will lead the way towards a more complex task next week.

During the previous lab session, I noticed that some students were unable to appreciate the importance of different Python Error types that the Python compiler provide. Being able to understand the errors that appear when you run your code is crucial for diagnosing and resolving issues efficiently. As you work on this exercise, you may encounter various types of errors. You are expected to be able to recognize and fix them by consulting the internet or using your critical thinking skills.

This URL may also serve as a helpful introduction to error handling. **Click on the underlined text to be redirected to the associated website**

Getting Started

To begin, download the `figurative.py` and `mexican.txt` files from the module page. Make sure files are located in a directory that can be accessed by the Python compiler. The `figurative.py` file contains a very simple stub program.

Ensure that you can run the `.py`, If you encounter an error such as

```
ImportError: No module named nltk
```

you should verify that the required library is installed. You can install it using pip by running the command

```
pip install nltk
```

or by following the appropriate steps in your IDE.

The program should print out the text of the following story:

> *Hunter wanted to be influential and powerful. So, Hunter kidnapped Princess and went to Chapultepec forest. Hunter's plan was to ask for an important amount of cacauatl (cacao beans) and quetzalli (quetzal) feathers to liberate Princess. Farmer thoroughly observed Hunter. Then, Farmer took a dagger, jumped towards Hunter and attacked Hunter. Suddenly, Farmer and Hunter were involved in a violent fight. Hunter went in search of some medical plants and cured Princess. As a result Princess was very grateful to Hunter. Hunter and Princess went to the Great Tenochtitlan city.*

This story was created using Mexica, an AI story writing system that produces tales in the style of traditional Mexican folklore.

If you're interested, you can learn more about this system by visiting this link: http://www.rafaelperezyperez.com/profile/publications/.

However, understanding this information is not necessary to complete this week's tasks.

In the above story, the adjectives are highlighted in red font. Specifically, they are:

- influential
- powerful
- cacao
- important
- violent
- medical
- grateful
- Great

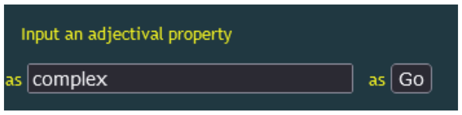
Tasks

The main objective of today's task is to utilize a web-based language engine called the Jigsaw Bard to enhance the story's language by replacing or adding figurative language to the existing adjectives (highlighted in red font).

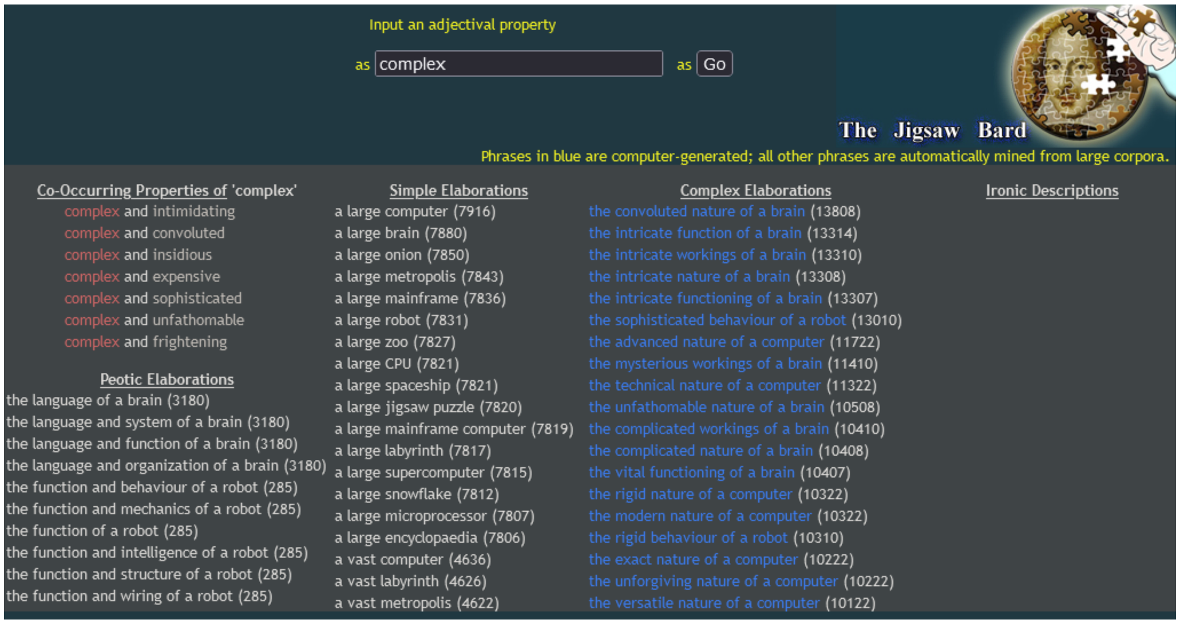
To begin, visit

http://bonnat.ucd.ie/jigsaw/

and input a word such as "complex" into the text box.



This will generate a list of metaphorical expressions:



Phrases in blue are computer-generated; all other phrases are automatically mined from large corpora.

Co-Occurring Properties of 'complex'
complex and intimidating
complex and convoluted
complex and insidious
complex and expensive
complex and sophisticated
complex and unfathomable
complex and frightening

Peotic Elaborations
the language of a brain (3180)
the language and system of a brain (3180)
the language and function of a brain (3180)
the language and organization of a brain (3180)
the function and behaviour of a robot (285)
the function and mechanics of a robot (285)
the function of a robot (285)
the function and intelligence of a robot (285)
the function and structure of a robot (285)
the function and wiring of a robot (285)

Simple Elaborations
a large computer (7916)
a large brain (7880)
a large onion (7850)
a large metropolis (7843)
a large mainframe (7836)
a large robot (7831)
a large zoo (7827)
a large CPU (7821)
a large spaceship (7821)
a large jigsaw puzzle (7820)
a large mainframe computer (7819)
a large labyrinth (7817)
a large supercomputer (7815)
a large snowflake (7812)
a large microprocessor (7807)
a large encyclopaedia (7806)
a vast computer (4636)
a vast labyrinth (4626)
a vast metropolis (4622)

Complex Elaborations
the convoluted nature of a brain (13808)
the intricate function of a brain (13314)
the intricate workings of a brain (13310)
the intricate nature of a brain (13308)
the intricate functioning of a brain (13307)
the sophisticated behaviour of a robot (13010)
the advanced nature of a computer (11722)
the mysterious workings of a brain (11410)
the technical nature of a computer (11322)
the unfathomable nature of a brain (10508)
the complicated workings of a brain (10410)
the complicated nature of a brain (10408)
the vital functioning of a brain (10407)
the rigid nature of a computer (10322)
the modern nature of a computer (10322)
the rigid behaviour of a robot (10310)
the exact nature of a computer (10222)
the unforgiving nature of a computer (10222)
the versatile nature of a computer (10122)

Ironic Descriptions

The Jigsaw Bard employs a collection of metaphors generated by a computer system that analyzes a vast corpus of text that identify nouns that are frequently described with the same adjective. You can refer to this paper for more information: https://www.aclweb.org/anthology/P11-4003.pdf.

We will use these metaphors to replace or enhance the existing adjectives in the story.

To achieve the goal, the following modification can be implemented:

1. Firstly, we must break the story into words or tokens, which involves dividing words such as "`don't`" into "`do`" and "`n't`".
   - We can use the `nltk` (Natural Language Toolkit) library's `nltk.word_tokenize` function to achieve this.
   - Update the `tokenize_and_pos_tag` method with the `nltk.word_tokenize` and assign the returned value to the variable `listOfWords`
   - Add a print statement to display the `listOfWords`.
   - Execute your code. If you have not previously used `nltk`, you will likely receive an error message that directs you to include specific lines in your code to download certain databases.
   - If necessary, adjust your code to address these error messages. Further information is available at http://www.nltk.org/data.html.
2. Next, we need to identify the adjectives in the story. This can be accomplished using a part-of-speech tagger, which assigns grammatical tags to each word in a text.
   - NLTK provides a built-in function called `pos_tag` that can be used for this purpose. You can find more information about this function at this url:
   - Take some time to understand the documentation and update the `tokenize_and_pos_tag` method to use the `pos_tag` function. Assign the resulting output to a variable named `taggedListOfWords`.
   - Add a line of code to print `taggedListOfWords`.
   - This will return a list of pairs (Python tuples), such as

```
[('Hunter', 'NNP'), ('wanted', 'VBD'), ('to', 'TO'), ('be', 'VB'),
('influential', 'JJ'), ('and', 'CC'), ('powerful', 'JJ'), ('.', '.'), ('So',
'RB'), (',', ','), ('Hunter', 'NNP'), ('kidnapped', 'VBD'), ('Princess',
'NNP'), ('and', 'CC'), ('went', 'VBD'), ('to', 'TO'), ('Chapultepec',
'NNP'), ('forest', 'JJS'), ('.', '.'), ('Hunter', 'NNP'), ("'s", 'POS'),
('plan', 'NN'), ('was', 'VBD'), ('to', 'TO'), ('ask', 'VB'), ('for', 'IN'),
('an', 'DT'), ('important', 'JJ'), ('amount', 'NN'), ('of', 'IN'),
('cacauatl', 'NN'), ('(', '('), ('cacao', 'JJ'), ('beans', 'NNS'), (')',
')')]
```

     where the first item is the word, and the second item is its grammatical tag.

   - If you would like to learn more about part-of-speech tagging, you can refer to this resource:
   - Make sure that the `tokenize_and_pos_tag` method returns the variable `taggedListOfWords`.

3. Optional: if you prefer to implement your own POS (part-of-speech) tagger,
   - You can download a list of adjectives from a source such as http://www.ashley-bovan.co.uk/words/partsofspeech.html.
   - Then, you can write code to compare each word in the story against the words in that list.
   - Note that this method is not very precise, as some words (such as "green") can have multiple parts of speech (in this case, "green" can be both an adjective and a noun).
4. The next step is to extract all the adjectives from the `taggedListOfWords`.
   - Adjectives are tagged with "`JJ`", for example, ('`grateful`', '`JJ`')
   - You can implement this in the `filter_adjectives` method using a list comprehension or a for loop. If you are a more advanced Python user, you may consider using a `lambda function` with the `filter` and `list` functions.
   - If you have written your code correctly so far, the resulting adjectives list should contain:

     ```
      ['influential', 'powerful', 'important', 'cacao', 'violent',
     'medical', 'grateful']
     ```
   - Make sure that the `filter_adjectives` method returns all the adjectives found.
5. Steps 3 and 4 can be combined into a single method called `add_figurative_language_terms_to_dict` as follows

```
def add_figurative_language_terms_to_dict(self, story):
    all_adj = self.filter_adjectives(self.tokenize_and_pos_tag(story))
```

6. Next, we will programmatically extract the metaphors and similes from the Jigsaw Bard that will be used to replace or supplement each of the adjectives in the story.

   i. First, we need to download the Metaphors and Similes from the Jigsaw Bard.
   - For example, let's consider the adjective "`complex`". We can use a programming approach to extract the metaphors and similes from the Jigsaw Bard by calling the following `URL`:

     "`http://bonnat.ucd.ie/jigsaw/index.jsp?q=complex`"
   - This can be done using the `requests` package. More details about the package are available via the following package documentation.
   - Add the following two lines under the method get_figurative_language_term:

     ```
     params = {'q': param}
     response = requests.get(self.url, params=params)
     ```

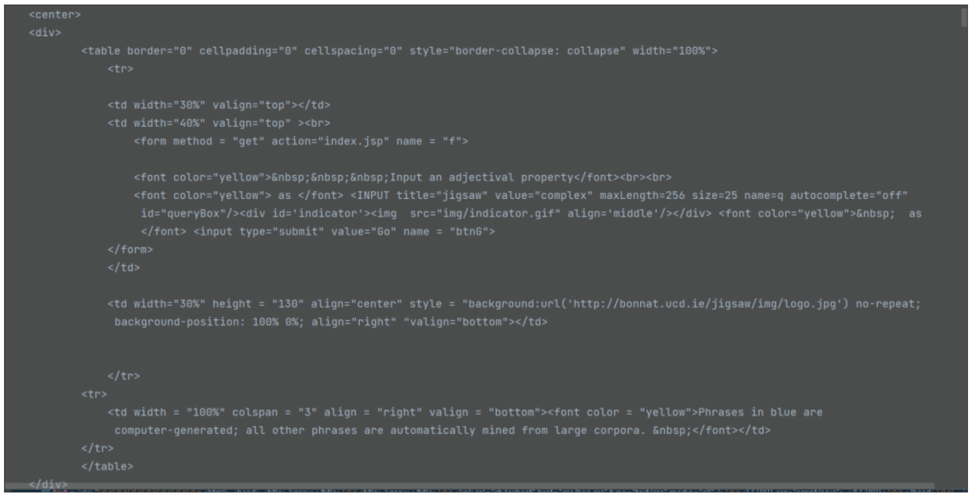
   In this scenario, we assign the `self.url` variable under the `__init__` method to '`http://bonnat.ucd.ie/jigsaw/index.jsp`'.

We will also pass in the `param` variable, which represents the adjective we are interested in extracting figurative language for. In our example, we assign `param` with the value '`complex`'.

- To see what the response input is, make the following changes under the `add_figurative_language_terms_to_dict` method, add:

```
self.get_figurative_language_term('complex')
```

- With the `PyCharm` debugging tool, pause the execution just after the response and click on it to inspect its contents. You should see...



```
<center>
<div>
        <table border="0" cellpadding="0" cellspacing="0" style="border-collapse: collapse" width="100%">
            <tr>

            <td width="30%" valign="top"></td>
            <td width="40%" valign="top" ><br>
                <form method = "get" action="index.jsp" name = "f">

                <font color="yellow">&nbsp;&nbsp;&nbsp;Input an adjectival property</font><br><br>
                <font color="yellow"> as </font> <INPUT title="jigsaw" value="complex" maxLength=256 size=25 name=q autocomplete="off"
                 id="queryBox"/><div id='indicator'><img  src="img/indicator.gif" align='middle'/></div> <font color="yellow">&nbsp;  as
                 </font> <input type="submit" value="Go" name = "btnG">
            </form>
            </td>

            <td width="30%" height = "130" align="center" style = "background:url('http://bonnat.ucd.ie/jigsaw/img/logo.jpg') no-repeat;
             background-position: 100% 0%; align="right" "valign="bottom"></td>


            </tr>
        <tr>
            <td width = "100%" colspan = "3" align = "right" valign = "bottom"><font color = "yellow">Phrases in blue are
             computer-generated; all other phrases are automatically mined from large corpora. &nbsp;</font></td>
        </tr>
        </table>
</div>
```

- If you look at the structure of the `HTML` file, you will see that all of the figurative expressions are encoded in the following form:

```
<center>
    <div style="width:100%;height:360;overflow-x:hidden;overflow-y:scroll">
        <table border="0">
            <tbody>
                <tr>
                    <td width="100%">
                        <a href="index.jsp?q=complex&amp;
                        longvehicle=a large computer">
                            <font style="--darkreader-inline-color: #dedcd9;"
                                  data-darkreader-inline-color=""
                                  color="black">a large computer
                            </font>
                        </a> (7916)
                    </td>
                </tr>
            </tbody>
        </table>
    </div>
</center>
```

In the simplified `HTML` structure presented above, the figurative expression "`a large computer`" is located within the `<table>` element and enclosed within `<a>` tags.

More detail about the tag `<table>` and `<a>` is available from the following link

- ➢ https://www.w3schools.com/tags/tag_table.asp
- ➢ https://www.w3schools.com/tags/tag_a.asp

- Alternatively, you can use the `'Inspect Element'` tool available in your browser to examine the HTML structure. `'Inspect Element'` is a powerful tool that enables developers or QAs to view and modify the appearance of a live web page by making temporary edits. These edits can be used to experiment with the behavior of HTML, CSS, and JavaScript elements. For Mozilla Firefox, you can access the 'Inspect Element' tool by selecting 'Inspect' from the browser's context menu. For other browsers, please refer to the respective browser's documentation for more information.

ii. Next, we need to scape the html. `Web scarping` is the process of collecting and parsing raw data from the We. There are many `Python Web Scraping Tools`, however, I personally prefer the Beautiful Soup package due to its simplicity in navigating, searching, and modifying a parse tree in `HTML` files. You may consider other web scarping tools which is summarise in this write-up:

- We parse the response from the `requests.get()` using Beautiful Soup as follow:

```
from bs4 import BeautifulSoup
soup_html = BeautifulSoup(response.text, 'html.parser')
```

I anticipate that you may encounter errors when completing this step. Please attempt to solve any issues on your own before seeking assistance. At the very least, try to identify the problem. If you are still unable to resolve the issue, please raise your hand for help.

- However, this still does not filter out all the figurative element. As explain in **6(i)**, the figurative expression "`a large computer`" is located within the `<table>` element and enclosed within `<a>` tags. You may consider the function `` `find_all` `` to filter the figurative element. More detail about `` `find_all` `` function is available via the package documentation:

- If you are able to run the code successfully, you should get a list of figurative elements as follow

```
['a large computer', 'a large brain', 'a large onion', 'a large
metropolis', 'a large mainframe', 'a large robot', 'a large zoo', 'a
large CPU', 'a large spaceship', 'a large jigsaw puzzle’]
```

Please note that the above list of strings represents only the top `10` figurative elements for the adjective '`complex`' due to space limitations. In fact, there are a total of `100` figurative elements for this adjective.

- A good way to store the results is in a Python dictionary, where the keys are the adjectives and the values are the lists of metaphors. For example:

```
{ 'influential': ['a powerful president', 'a powerful critic'],
'powerful' :[ 'a wealthy computer', 'a wealthy king'],
'important': ['a special CEO', 'a famous CEO']}
```

- Therefore, you need to do the following changes.
  i. Under the `add_figurative_language_terms_to_dict`

     Delete

     `self.get_figurative_language_term('complex')`

     replace with

     ```
     query_params = [tup[0] for tup in all_adj]


     figurative_language_terms =
     self.get_figurative_language_terms(query_params)
     ```

  ii. add the following under `get_figurative_language_terms method`. This is just a snippet, you need to do the necessary changes to the program working

     ```
           figurative_language_terms = {}


          # iterate over each query parameter
          figurative_language_terms[param] =
     self.get_figurative_language_term(param)
     ```

- However, please note that the metaphors are generated by the analysis of a large amount of text, and are not filtered for offensive language. As an extension, you might consider filtering these out.
- In my implementation, I combine the activity of retrieving the URL and scraping the HTML under the method `get_figurative_language_term`, which also returns the variable `figurative_language_terms`.

7. To replace the adjectives in the story with metaphorical expressions, the following steps can be considered:
   - First, `loop` through the tokenized copy of the story

     ```
     text_to_list=nltk.word_tokenize(story)
             for word in text_to_list:
     ```

   - Check `if` the token is an adjective.
   - For each adjective encountered, select one of the metaphorical expressions extracted from Jigsaw Bard at random, from the list of expressions associated with that adjective (i.e., `figurative_language_terms`).

- Note that you should first check if the list of expressions for each adjective is empty. If the list of metaphorical expressions extracted from Jigsaw Bard for a particular adjective is `empty`, it should be left as it is
- If the list is `not empty`, you can create a list index slicer to randomly select a position from the list of metaphorical expression. This sub-method can be placed under the method `rand_pos`, which will return the random position to index the adjective position. For example:

```
def rand_pos(self, max_val):
        return np.random.randint(low=0, high=max_val, size=10)[-1]
```

  Here, `max_val` is the size of the corresponding adjective list, and `rand_pos` will return the index position to be sliced.
- Remember to include linking words such as "`like`" or "`as`" to connect the adjective and the metaphorical expression.

  For example:

```
    f"{word} as {figurative_language_terms[word][lpost]}"
```

  `word` represents the adjective of interest, and `lpost` represents the index location to be sliced.
- You can use an `if-else` statement to ensure that only adjectives are replaced.
- The updated tokens and non-adjective tokens can be appended into a variable called `new_text`. For example:

```
new_text.append(word)
```

- It's worth noting that the metaphorical expressions are generated by analyzing a large amount of text and are not filtered for offensive language. As an extension, you might consider filtering out offensive expressions.

8. To turn the `list of tokens` back into a `string`, you can use the join method.

   The final story should look something like this, where the adjective in the story is highlighted in red font, and the added figurative language is highlighted in blue font:

   Hunter wanted to be influential as a senior business leader and powerful as the force and voice of a storm. So, Hunter kidnapped Princess and went to Chapultepec forest. Hunter's plan was to ask for an important as the role of a leader amount of cacauatl (cacao beans) and quetzalli (quetzal) feathers to liberate Princess. Farmer thoroughly observed Hunter. Then, Farmer took a dagger, jumped towards Hunter and attacked Hunter. Suddenly, Farmer and Hunter were involved in a violent as a storm's immediate aftermath fight. Hunter went in search of some medical plants and cured Princess. As a result Princess was very grateful as an eager dog to Hunter. Hunter and Princess went to the Great Tenochtitlan city.

Extensions

Here are some extensions to the tasks above. While I do not expect you to attempt these during the lab session, they can provide ideas that may serve as the basis for your coursework.

1. Use a thesaurus such as Thesaurus Rex (http://ngrams.ucd.ie/therex3) to substitute for repeated words. For example, if the word "dagger" is used twice in the story, use the thesaurus to substitute a word like "weapon" or "blade" the next time. Use the same tool to add, perhaps randomly or in some thematically-relevant way, adjectives to the raw nouns in the story ("dagger" to "sacred dagger" for example). This can return an xml file so there is less fossicking around with scraping the html needed here when compared with the earlier example.

   Example syntax:

   http://ngrams.ucd.ie/therex2/common-nouns/member.action?member=dagger&kw=dagger&needDisamb=true&xml=true

2. Use the ideas in this worksheet to elaborate the language used in a chatbot program such as Eliza:

   https://github.com/jezhiggins/eliza.py/blob/main/eliza.py

3. (difficult but very interesting; could be a good basis for the coursework). Write your own metaphor-creation system. Start with the idea of searching a corpus of text for nouns that are commonly described with the same adjective. Then, use a thesaurus system to find other adjectives with similar meaning. For example, if "silent night" is a common adjective, then "Farmer was quiet" turns into "Farmer was quiet as night" via the intermediate, unused bridging adjective "silent".