POEM GENERATION

- **Github Project Link:** https://github.com/Shaherin/Poem_Generator

Members:
Shaherin Dehaloo
Sashni Pather
Jenade Moodley
Danielle Nikkita Naidoo

Introduction:

Writing poems has always been a challenging task as it involves both creativity and inspiration. Research has been done in various fields such as artificial intelligence and natural language processing to be able to master the various techniques for text processing and to be able to use them in order to achieve some aim.

Area of Application

This approach to poem generation relies heavily on information retrieval. A large part of this project involved fetching desirable sentences from a corpus, and processing these sentences. Poem generation does not necessarily require NLP. Pure statistic and probabilistic models can be used relatively successfully, however applying NLP techniques in isolation, or in conjunction with these models, will provide an objective improvement of the results.

Objectives:

From a corpus of poems, use NLP techniques to formulate a new poem and evaluate the result and performance of the system.

Justification of Certain Design Decisions

Miscellaneous short discussions.

•On the Change of Topic

The topic of the original project proposal was song generation. The decision to change to poem generation was based on the fact that coherent poems were easier to find i.e. less colloquialism was used, as well as less interjections and repetition. Poems usually followed their respective subjects more closely, were shorter, and the structure of poems was more well defined.

•On the Limiting of Corpus Poem Subjects

The idea to limit the subject of poems contained in the corpus was to remove one layer of variability from the corpus. It should be noted, however, that determining the subject of poems can be done (with varying degrees of accuracy) using various NLP techniques, and this was an intentional omission due to the difficulty prospect.

•On Concurrency

As stated above, this project falls under information retrieval/qualification. NLP procedures are also notoriously time consuming. Due to the read-only nature of the tasks required to be executed (therefore lack of deadlock possibilities if correctly implemented), and the number of independent tasks required to be executed to generate a single line of the poem; concurrency was decided to be a worthwhile time investment.
The introduction of concurrency into the program produced noticeable improvements in execution time on two separate systems. Concrete results and comparisons were deemed irrelevant and not recorded.
The effect and areas of usage will be discussed further as part of the discussion on class structure.

•On the use of Singletons

Singletons were overused in this particular project for investigation purposes in a multithreaded environment. Unjustifiable singletons were used for the Poem_Generator and Corpus classes, and these can easily be removed however they currently have no negative implications due to the small scale and relative simplicity of the project.

The justifiable singleton implementations are those of the Stanford_Wrapper, and WordNet_Wrapper classes. The decision to choose singletons stems from the fact that WordNet is accessed via a JWI object, where it is only necessary to have a single instance of this object. The Stanford pipeline initialization is a lengthy process, and initializes multiple modules, so it is fairly reasonable limit to a single instantiation. Singletons can almost always be replaced, and in this case their primary objective is to convey the need for singular instances which should not be statically instantiated.
All singletons were implemented as non-static, synchronized Meyer's singletons which are thread safe.

No other design patterns (or anti-patterns) were used, although others were considered appropriate, due to time constraints and inexperience. One example is that the Poem_Generator class is a prime candidate for the strategy design pattern.

An example of a Meyer's singleton's lazy initialization:

```
//private instance
private static WordNet_Wrapper wordNet;

//get instance
public static synchronized WordNet_Wrapper getInstance()
{
    //synchronized method ensures that no two threads enter this "if" block
    if(wordNet == null)
    {
        wordNet = new WordNet_Wrapper();
    }
    return wordNet;
}
```

•On Language, APIs, and Software Reuse

Poems rely heavily on parts of speech, and the WordNet database provides just this. The Java WordNet Interface (JWI) was used as an interface between our program and the WordNet 3.1 database, which will be discussed further under class structure.

The Stanford CoreNLP Library (2015) was used for various NLP techniques.

Rhyming words are retrieved via the online service rhymebrain.com.

The project was coded in Java 8 due to the need (or rather preference) for lambda expressions in relation to concurrency, and the dependency of certain modules of this particular version of the Stanford CoreNLP Library.

Achievements

The generator is able to:

- Search the corpus on multiple threads
- Use regex, Stanford CoreNLP, and WordNet functions to select certain types of sentences and process them
- Get sentences that contain: subject, word, rhyme, random sentence
- Execute batches of WordNet functions asynchronously
- Tokenizing, OpenIE relation triples, POS tagging(using Stanford's PTB Tagger)
- Definitions, holonyms, hypernyms, meronyms, synonyms, stemming, rhyming words are retrievable for a given word
- Syllable counting for sonnets
- WordNet can be loaded directly into memory for runtime performance boosts

- Generate free verse poems(user defined structure) or sonnets (fixed structure)

Shortcomings

- The main shortcoming of the program is the poem generation algorithms themselves. Many WordNet and Stanford CoreNLP functions were written and not used due to time constraints, all of which would contribute to the quality of the poems.
- Exceptions still occur on some sentence fetching actions despite all current safety measures(the program could be more polished in general)
- More research could have been done to ease the burden of the generation algorithms

Class Structure

• Corpus

The corpus class contains methods for loading the corpus, and searching the corpus.
The corpus is loaded into four Strings, with the intention of each string being searched on a separate thread to greatly reduce searching time. This class contains one of two instances of Executor_Wrapper present throughout the project to invoke the multiple searches.
The corpus contains functions for returning specific types of sentences, specified by an enumerated type. The types are namely subject, word, rhyme, and random.
These sentence fetching functions are implemented with both String and Callable return types, which are both required depending on the context.

Multithreaded sentence fetching functions return Callables using lambda expressions to be invoked later.

```
/** Multithreading functions */
  public Callable<String> getSentence_Callable(SentenceType type, String word)
  {
      return () -> {
          return getSentence(type, word);
      };
  }

  public Callable<String> getSentence_Callable(SentenceType type)
  {
      return () -> {
          return getSentence(type);
      };
  }
```

String version of sentence fetching (sentence containing subject)

```
case CONTAINS_SUBJECT:
{
    //get subject with stanford_tools
    //here we are required to pass a sentence as String word for this to work
    String[] relation_triple = stanford_tools.getRelationTriple(word);
    String subject = relation_triple[0];

    //find a word in sentence
    Pattern pattern = Pattern.compile("(\\b"+ subject +"\\b)");

    //search corpus for sentences that match pattern
    List<Callable<ArrayList<String>>> callables = searchCorpus(pattern);
    sentences = executor.invokeAll(callables);

    if(!sentences.isEmpty())
    {
        //choose a random sentence //((max - min) + 1) + min
        int index = random.nextInt(sentences.size());
        final_sentence = sentences.get(index);

        return final_sentence;
    }
    break;
}
```

Multithreaded search function. Search is done based on a regex pattern

```
/* -Corpus is searched using 4 threads each searching a quarter of the corpus
 * -returns list of 4 callables
 */
public List<Callable<ArrayList<String>>> searchCorpus(Pattern pattern)
{
    List<Callable<ArrayList<String>>> callables = new ArrayList<Callable<ArrayList<String>>>();

    Callable<ArrayList<String>> search1 = () -> {
        return searchCorpus(pattern, corpus_string_1);
    };

    Callable<ArrayList<String>> search2 = () -> {
        return searchCorpus(pattern, corpus_string_2);
    };

    Callable<ArrayList<String>> search3 = () -> {
        return searchCorpus(pattern, corpus_string_3);
    };

    Callable<ArrayList<String>> search4 = () -> {
        return searchCorpus(pattern, corpus_string_4);
    };//*/

    callables = Arrays.asList(search1, search2, search3, search4);
    //callables = Arrays.asList(search1);

    return callables;
}
```

• Stanford_Wrapper

The Stanford_Wrapper class wraps all Stanford related functions, and handles the initialization of the Stanford pipeline. This class is implemented as a singleton, for reasons mentioned in the earlier discussions.
The class contains OpenIE relation triples, POS tagging, and tokenizing.
It is currently useable, however there is much that was omitted due to time constraints. Callable versions of all functions were not implemented, and the functionality of the class as a whole could be improved. The Stanford CoreNLP library requires great time investment to use correctly and effectively; this is the reason for the seemingly limited nature of this class.

Constructor: initialization of pipeline and POS tagging model

```
private Stanford_Wrapper()
{
    // Create the Stanford CoreNLP pipeline
    tagger = new MaxentTagger("stanford-postagger-full-2015-12-09//models//english-caseless-left3words-distsim.tagger");

    Properties props = new Properties();
    props.setProperty("annotators", "tokenize,ssplit,pos,lemma,depparse,natlog,openie");
    pipeline = new StanfordCoreNLP(props);

}
```

From the Javadoc generated for this project:

Modifier and Type	Method and Description
static Stanford_Wrapper	getInstance()
java.util.concurrent.Callable<java.lang.String[]>	getRelationTriple_Callable(java.lang.String sentence) Multithreading functions
java.lang.String[]	getRelationTriple(java.lang.String line) Stanford Functions
java.lang.String	POS_Tagger(java.lang.String line)
void	printRelationTriple(edu.stanford.nlp.ie.util.RelationTriple triple) Print Functions
java.util.ArrayList<java.lang.String>	tokenize(java.lang.String line)

• WordNet_Wrapper

This class wraps access to the WordNet database via the JWI. Access to the database is via a RAMDictionary object, which has the option to be loaded into memory for faster runtime access. This option was provided in the class.
The class contains functions for retrieving holonyms, hypernyms, hyponyms, meronyms, synomyms, word stems (which are crucial for searching the database), and rhyming words, which are retrieved via URL connection to rhymebrain.com. This is not related to WordNet, however its functionality is similar. It was therefore decided that it's inclusion in this class was logical if not entirely correct.
The class contains multithreaded versions of all these functions which return Callables.

The Javadoc entry for this class is lengthy and shall be omitted, as all functions are described above.

• Executor_Wrapper

The Executor_Wrapper wraps a java ExecutorService object, which handles invoking of multiple Callibles/Runnables on multiple asynchronous threads.
The ExecutorService is implemented as a CachedThreadPool, which creates threads as needed. A performance improvement could be introduced in the form of a FixedThreadPool, as the max number of threads executed by this program per cycle can be determined.
The function of this class is to hide the procedure of retrieving Futures from an invoked list of Callables (this project exclusively uses Callables as returns from threads are always required), and to provide a safe executor shutdown which will wait for completion of all threads.
This class invokes has functions to invoke List<Callable<ArrayList<String>>>, List<Callable<String>>, and generic type Callable<T>. The invoke functions will then return the appropriate results, abstracting the retrieval of the results from Futures.

From the Javadoc:

Modifier and Type	Method and Description
java.util.List<java.lang.String>	invokeAll_String(java.util.List<java.util.concurrent.Callable<java.lang.String>> callables)
java.util.List<java.lang.String>	invokeAll(java.util.List<java.util.concurrent.Callable<java.util.ArrayList<java.lang.String>>> callables) Functions
void	Shutdown_Executor()
<T> T	submit(java.util.concurrent.Callable<T> callable)

• Poem_Generator

This is the heart of the program, and uses all other above mentioned classes. It contains the second instantiation of Executor_Wrapper, to invoke multiple calls to the WordNet and Stanford wrapper classes. Pattern matcher pairs are used consistently in this class.
This class contains functions to generate free verse and sonnets. The basic structure of the algorithms are shown below in pseudo code:

-Free Verse:

```
FOR (each stanza)
   WHILE (first line is not acceptable)
      Find random sentence

     IF (random sentence does not begin with conjunction)
       First line is acceptable
     ENDIF
  END WHILE

Add first line

FOR (each line in stanza)
    IF (line is odd)
      Get random sentence
    ELSE
      Get word at the end of the last line of the poem
      Get words that rhyme with this word
    ENDIF

    IF(no rhyming words are found)
      Add random sentence
      Continue
   ENDIF

 Search corpus for sentences that end with words in this list
 Select one of these sentences
 Add sentence

 IF(current line = last line of stanza)
    Check if line ends with mid-sentence punctuation and remove it
    Check if line ends with determiner and remove it
 ENDIF

 ENDFOR
ENDFOR
```

-Sonnet (written by Sashni Pather)

The generateSonnet method returns an arraylist of strings consisting of the lines of the poem. This method selects two random starting sentences of 10 syllables to be used as the first two lines in the first stanza of the poem. The end word of the first line is then passed through to rhymebrain.com which returns an arraylist of rhyming words. A search of corpus is done and sentences ending in any of these rhyming words are stored in a list. A randomized index chooses which sentence should be line 3 in the poem. It continues in this fashion to generate line 4 of the poem. (This will also be implemented in the generation of stanza 2 and 3 and the rhyming couplet.)
To count the syllables of a sentence. It is broken down into words and each word is thereafter testedby certain rules to count the number of syllables in it.

• Bag of words Statistical Search - not implemented (written by Jenade Moodley)

BagOfWords:

Method returns a bag of words (words related to the subject) in the form on an arraylist.
Method takes in the subject and the synonyms of the subject as parameters. Subject and synonyms are added to the arraylist. Method traverses the corpus, one line at a time, and checks for sentences that contain the subject word. Two words prior and two words after the subject word are added to the arraylist.
Looped until end of corpus. (Appropriate checks are done so as no word appear twice in the arraylist). Same procedure is followed for each synonym. Arraylist is then run through the stanford API to remove determinants. The arraylist is returned.

EndWords:

Method takes in an arraylist of rhyming words for the end term. This arraylist is thus filled with "end words". An arraylist containing all the verses with the matching end term is returned
The arraylist is of type Verse. The Verse class structure allows us to store a verse from a poem as well as which poem it is from. Method then takes this arraylist, traverses the corpus, one line at a time, and checks for sentences that end with each end word. If loop reaches a "#", it is a signal that it is the start of a new poem If loop finds a sentence that ends with the corresponding end word, the verse and the poem number is added to the arraylist.
Loop continues until end of arraylist of end words. Appropriate checks ensure no poem title is added to the arraylist. Arraylist is returned.

verseWithBagOfWords:

Method takes in an arraylist of type Verse which contain sentences with the specified end words (worked out by
the EndWords method) and an arraylist containing the bag of words.
Each verse is then compared with all the words from the bag of words and will only be chosen if that verse
contain at least one word from the bag of words.
If a verse is chosen, the verse and the song it is from is added to an arraylist of type verse.
Loop until end of arraylist of verses.
The arraylist is returned.

Evaluation Technique

Evaluation of the results was done using a test derived from Turing's Test, where test subjects were made to detect generated poems amongst a list containing both generated and written poetry. Other techniques were considered, such as evaluating the poems using another NLP library. It was decided that this would have logically reduced to another NLP library evaluating the results of the Stanford CoreNLP library (which was used in this project) – effectively circumventing our result.
The chosen method is admittedly not perfect and could be improved in a number of ways; as discussed below.

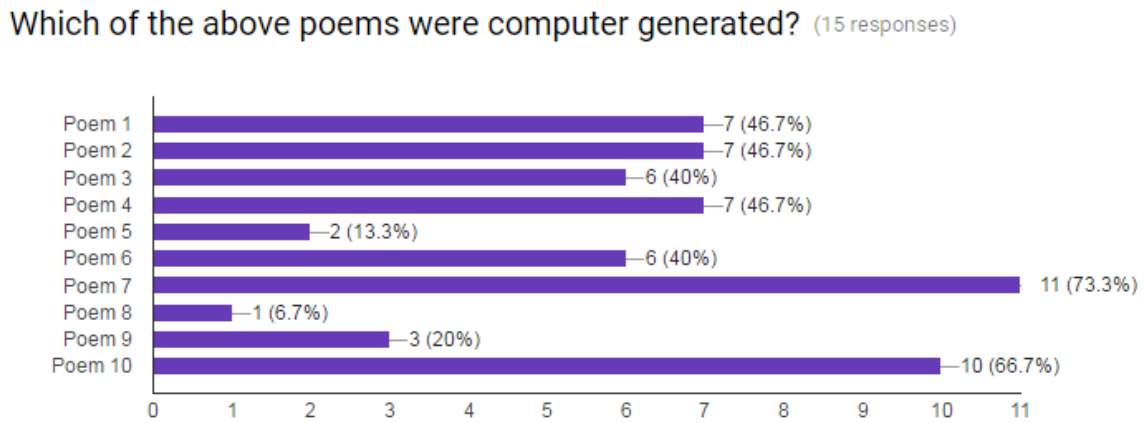
Two surveys were conducted with each containing ten poems, four of which were generated. The candidates were allowed only four guesses. A point was awarded for a correct guess, and a point deducted for an incorrect guess. This was done as incorrect guesses indicate a point awarded to one of the successful poems. It also indicates uncertainty in the candidates and essentially works towards averaging the quality of the generated poems contained in the list.

Is this a fair test? No. The results are skewed in our favour by at least five counts:

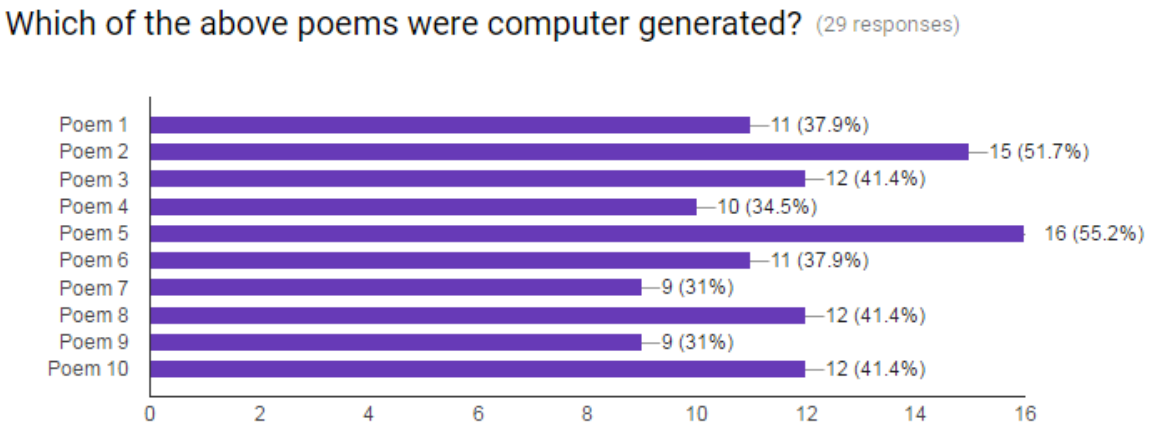
1. The generated poems are embedded among a higher number of written poems.
2. The negative marking punishes guesses, which is a shortcoming of the Turing Test, however (considering 1) this is much to our advantage.
3. The candidates were volunteers and are not necessarily qualified to judge poetry on any level.
4. The cryptic nature of poetry itself obscures the fact that the generated poems often do not make ense but may pass the test nonetheless.
5. The generated poems (in Test 2 specifically) were chosen specially for the test and do not necessarily represent the average quality of poems generated, where quality may also be a subjective measure.

The results will be shown and discussed below.

Test 1:
https://docs.google.com/forms/d/1cVzoCUykBWqVIXMW1T3r-v_w9PlBphRoTBOkQr_7y9M/edit?usp=forms_home&ths=true



Test 2:
https://docs.google.com/forms/d/1SEk_8aFKS4FB2cbQwepUcKKx_NU04kLYyBt2vYHiHCY/edit?usp=forms_home&ths=true



Solutions:
Test 1: *3, 4, 7, 10* ***Test 2:*** *2, 3, 6, 8*
Following the scoring system indicated in the tests (test 2 being the subjectively more difficult test), the collective user scores are as follows:
Test 1: +8
Test 2: -17

The results for test 1 indicate that most users were able to identify more of the generated poems.
The results for test 2 were fairly uniform, indicating a great deal of uncertainty among test subjects.

Relevant Links

Github Project: https://github.com/Shaherin/Poem_Generator
JWI: http://projects.csail.mit.edu/jwi/
Stanford CoreNLP: http://stanfordnlp.github.io/CoreNLP/
Rhymebrain: http://rhymebrain.com/en