EECE 2140 Project Proposal

Chance Bowman

16685

Application Description

* The interactive application is a spell checker, which aims to read input text files and suggest modifications for said files. There is a current intention to, additionally, implement a graphical user-interface, highlighting the possible places for modification.
* The application will check for text in a given file and compare each word to a dictionary of known words. If a given word is not found, it will propose a small number of similar words. The user will then have the option to either accept one of said similar words, suggest their own change, or ignore the discrepancy. The application will also aim to bring attention to unknown words, through the use of highlighting. While there are multiple interaction loops present in the application, it would appear to one that, technically, it’s a single interaction loop. The user will input a file, then subsequently choose a file type for which the application will check.
* The application will accept inputs from the command line, pulling from the user’s local database. The type of the input may be either plain-text, HTML, or PDF.
* The application will produce both text-based and graphical outputs. It will print suggestions for unknown words, which the user may choose from, if deemed appropriate. It will also produce a visual output, highlighting areas of text that contain said unknown words. Its primary function is file modification, which is optional, at best.

Application Design

OOP

* Spellchecker(); checks the given input file for unknown words, giving the user options for modification
  + \_\_init\_\_(self, ?); initializes class (with a possible instance)
  + spell\_check(self, ReferenceFile); begins spellchecking a given file, calling on methods to parse the file and check words, based on input type
* ReferenceFile(); an abstract class, which serves as a blueprint for the more specific input type classes (will be called upon in each of the subclasses)
  + text; the input file
  + \_\_init\_\_(self,text); initializes class with input file text
  + parse(self); parses the input file, returning a list of comma-separated words
* Suggester(); presents to the user a list of suggestions to replace unknown words in the input file
  + known\_words; a list/set containing known words (correctly-spelled, from dictionary)
  + \_\_init\_\_(self,known\_words); initializes class with a list/set of known words
  + checker(self,word); checks to see if a given word is contained in known\_words
  + suggestions(self,word); if a word is not found is known\_words, it will give suggestions from known\_words
* TextCheck(); calls on ReferenceFile(), and checks/parses plain-text files
  + text; will be inherited from ReferenceFile and represent text from plain-text file
  + parse(self); will parse text from plain-text file, returning a list of comma-separated words
* HTMLCheck(); calls on ReferenceFile(), and checks/parses HTML files
  + text; will be inherited from ReferenceFile and represent text from HTML file
  + parse(self); will parse text from HTML file, returning a list of comma-separated words
* PDFCheck(); calls on ReferenceFile(), and checks/parses PDF files
  + text; will be inherited from ReferenceFile and represent text from PDF file
  + parse(self); will parse text from PDF file, returning a list of comma-separated words
* GraphicalInterface(); will highlight unknown words \*\*\*need to research\*\*\*
  + \_\_init\_\_(self); initializes the graphical user-interface

------------------------------------------------------

* Spellchecker() “has a” relationship with Suggester()
* GraphicalInterface() “may have a” relationship with Spellchecker()
* ReferenceFile() “is a” parent class of TextCheck(), HTMLCheck(), and PDFCheck()

------------------------------------------------------

* To satisfy the requirement of extensibility, I will have a parent class, named ReferenceFile, which will serve as the general blueprint for the subclasses TextCheck, HTMLCheck, and PDFCheck. For instance, in TextCheck, all of the plain-text will be parsed and checked against the known\_words. On the other hand, in HTMLCheck, all 'real' content (no tags) will be parsed and checked against the known\_words.

------------------------------------------------------

Library Support

* HTML Parsing: BeautifulSoup / lxml
  + Will be used to parse HTML content in HTMLCheck.
* PDF Parsing: PyPDF2
  + Will be used to parse PDF content in PDFCheck.
* Graphical User-Interface: Tkinter
  + Will be used to create a graphical user-interface.
* Unit Testing: unittest
  + Will use to write test cases for code.
* Command Line Arguments: sys
  + Will be used to handle command line arguments, with file specification.
* \*\*\* All else, it is believed, will be implemented from scratch. \*\*\*

------------------------------------------------------

Testing

* The unittest library will be used to test individual methods and functions, extensively. Other testing may be done through the use of the debugging tool (this includes interactions between classes, themselves, and parsers). It currently appears as though automated testing for the graphical user-interface may prove difficult. A common problem with automated testing, which will be found in the project, will be input variation (the scope of possibilities are extremely wide). To make the code most easily testable, it may be in interest to separate certain parts, based on their components to the code, overall. In this regard, one may test as-they-go, building a strong foundation to grow from.

Timeline

* November 27:
  + Implement basic algorithm for spellchecking (not yet considering extensibility)
  + Create a command-line interface for the application
  + Develop unit tests for components of basic algorithm
* December 4/6:
  + \*\*\*May move some of the former, here, depending on unexpected events over Thanksgiving Break\*\*\*
  + Have the Levenshtein distance calculation integrated into the application
  + Try to have implemented extensibility for HTML/PDF
  + Possibly have the GUI set-up
* December 11:
  + Finish extensibility, if not yet done
  + Finish the GUI
  + Finish writing unit tests
  + Write DOCSTRINGS
  + Possibly, also integrate with symmetric spelling correction (SymSpell)