Molly Yang, Jose Castillo, Alex Chen, Shannon Lytle
CS51 Final Project Final Specifications
4/12/13

Signatures/Interfaces:
Data structure: trie

autocorrect

distance (string, int, trie, bool) :

levenshtein (string, string) : int

mostProbable (string, int) list : string

/* autocorrect is the grand compiler function*/
autocorrect : string

autocomplete
autocomplete (string) : string

other
scrape (file, trie) : trie
saveTrie (trie) : file
opentTrie (file) : trie

Main algorithm:

distance returns all (word, word probability) that are <= maxDistance edits from word1
distance (str : word1) (int : maxDistance) (trie : wordTrie) (bool : iseow): returns (word, probability) list
- traverses through trie node by node
- at each node, calculate levenshtein distance between word1 and the word as of the node
- stores that distance in a matrix
- return the bottom right distance of the full matrix
- if that distance <= maxDistance, then add (word, word probability) to list
  - if distance is called with iseow true, then only complete words will be in this list
  - else, any sequence of nodes, not necessarily a complete English word, will be in

this list
return full list of (word, probability)

levenshtein calculates the minimum number of edits to change word1 to word2
levenshtein (str: word1) (str: word2)
min of:
cost: if word1[x] == word2[x], then cost = 1
deletes : 1 + levenshtein word1[0-2nd to last char], word2
inserts : 1 + levenshtein word1, word2[0-2nd to last char]
substitutions: cost + levenshtein word1[0-2nd to last char], word2[0-2nd to last char]
return min value

mostProbable takes the (word, probability) list returned by distance, and outputs the most probable word
mostProbable ((str : word), (float : word probability)) list: returns (str: word)
iterate through list Word1::Word2::tail
compare Word2 with Word1 and Word3:
if Word2 prob < bayes prob, then mostProbable Word1::tail
else if Word2 prob > Word1 prob, then mostProbable Word2::tail
else if Word2 prob = Word1 prob, then Word1::(mostProbable Word2::tail)
if there is more than one word at the end, use random number generator to pick
arbitrary word
else if there's only word word, return that
returns that word

autocorrect
mostProbable (distance word1 2 trie true)

autocomplete query trie activenodes
if the list of active nodes is empty,
create list of active nodes with distance word1 2 trie false
else update list of active nodes based on new letter [may split off into own function]
for each active node
traverse the trie, at each point going towards the node with the highest count
return the most probable word for each active node, append to list of words
return mostProbable wordlist

scrape (file: txt) (trie: t)
initializes trie (could be empty)
read txt into memory

```
        split txt into words by tokens (punctuation, spaces)
        for each word
                check if word exists in trie
                        if yes, update counts for the nodes of each letter in the word
                        and update counts for the final node
                else
                        create nodes as needed, initializing them with counts of 1
                        update existing nodes' counts
        return trie with updated counts

data structure interface

Trie node signature
Class Node ():
        char /* character that this node represents */
        string /* string that traversal to this node represents */
        count /* count of words that traverse this node in any way*/
        counteow /* count of word that ends in this node */

/* return a word or partial word */
def get_string():
        return string

/* return the next node given a an edge specified by a char */
def get_next_node(char):
        return node

/* create the next node given an edge specified by a char */
def set_next_node(char, node):
        return node

Trie signature
Class Trie():

Outputs a trie data structure from a list of words that groups common prefixes
def make_trie(*words):
        return trie

/* naive implementation of tries takes in a list of word strings
 * creates nodes and returns a completed trie */
def __init__(words):
        return trie
```

```
Takes in a trie and word to lookup
def in_trie(trie, word):
        return bool

return trie with word added
def insert (trie, word):
        return trie

return trie with word removed
def remove (trie, word):
        return trie
```

Timeline:

Week 1:

- Scrape text file
- Implement data structure
- Have autocorrect algorithm mostly finished
- Store trie in memory

Week 2:

- Finish autocorrect algorithm
- Have autocomplete partly finished, i.e. without the letter-by-letter active node updating
- Improve efficiency of all written parts
    - consider switching to damerau-levenshtein distance
- Improve data structure

Week 3:

- Finish/debug auto-complete algorithm
- Potentially work on word-learning extension
- Potentially work on additional extensions
- Debug all
- Potentially write interactive interface

Progress Report:

Because the way in which our algorithm needs to be implemented is very front-heavy, we will be distributing the work of our first week to the first week and a half so that the work for the entire project is more evenly distributed throughout the three weeks. This gives us additional time to refine and debug our auto-correct algorithm, which is the bulk of the project. By the end of week one, an autocorrect infrastructure will exist, and by the middle of week two, it will completely work. Then, the autocomplete component of the algorithm will begin to be implemented, with the

last week dedicated to extensions.