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

Brief Overview
Our team seeks to create a text-prediction and auto-correct algorithm. When drafting texts, emails, etc., one may find it cumbersome to type an entire word, phrase, or sentence. Our tool addresses this problem by autocompleting and spell-checking texts, emails, and other written works. This makes the process of writing (e.g. texting, emailing) faster, while producing more accurate text. Our goal for this project is to create an algorithm that can autocomplete a word or phrase with the most likely suggestions and auto correct misspellings.

Feature List
Core features (first checkpoint):
1. Implement data structure that stores words
2. Scrape long texts to determine how commonly different words are used
3. Algorithm that autocorrects a word (structure: http://norvig.com/spell-correct.html)

Cool extensions (second checkpoint):
3. Algorithm that returns a suggestion for the most likely word to complete a partially typed word based on that partially typed input

Potential additional extensions (final turn-in):
5. Use input text to change probability of word
6. Learn new words

How work will be split
Jose will implement the data structure (for first checkpoint)
Molly will implement the autocorrect algorithm (for first checkpoint)
Alex will implement the scraper/parser to determine common-ness of word, and using input text to change relative common-ness of word (for first checkpoint)
Alex or Shannon will implement the auto-complete algorithm (for second checkpoint)
If there’s time left, someone will implement the function that allows new words to be learned.

Technical Specification
We will create an interface to represent the data structure (and correspondingly, how to extract and input data into that data structure) that stores the words, along with their relative likelihood of being the auto completed word.

Modularization:
Modularization of the data structure is described below. (I’ve never coded in Python, so my signatures are

pseudo-code)

Autocorrect:
Scrape will take a big text file (as explained below) to build the trie and denote the counter (number of times the word appears)

The advantage of using the trie implementation as opposed to the hash table implementation (as dictated by norvig.com) is that the determination of the possible edits and their status as a "known" word is done simultaneously. However, determination of the possible edits is much trickier in tries.

edit takes in a word and transverses the tree to calculate the following functions

```
splits
deletes
transposes
replaces
inserts
return deletes + transposes + replaces + inserts
```

If we have time, we would also define a function **edit2** which would increase the accuracy of our spell checker. (Again, this is more complicated than simply calling edit twice as norvig.com does, given the nature of tries)

correct which calls edit on the word and takes the word with the max given probability.

Autocomplete (cool feature):
In the naive implementation, we have a function **autocomplete** which takes in the given prefix, traverses the tree, and returns the word with the max probability. We also have **inc** which calls this function incrementally as the prefix is changed. In the extended fuzzy search implementation, we would have another **determine** which determines the active nodes given the prefix (autocomplete is called on the results of determine).

Algorithm (Main)

1) We will use the Levenshtein Distance algorithm and Bayes theorem to create an algorithm that auto-corrects a word. Our suggested word is the one with the maximum probability of a correction suggestion given the original word. IE
Max Prob(Original Word | Corrected Suggestion) * Prob(Corrected Suggestion)/Prob(Original Word), or analogously,
Max Prob(Original Word | Corrected Suggestion) * Prob(Corrected Suggestion)

http://stevehanov.ca/blog/index.php?id=114

We will take words out from our word data structure (how to do so will be detailed in the data structure section below), and use that to calculate this probability model. The probability of that word will be stored in the data structure (after the parsing is done).

This algorithm will be applied to every word/part of word that the user inputs.

2) We will use a second algorithm that autocompletes a partially typed word:

We will attempt to implement autocomplete by roughly the same method detailed in http://www.ics.uci.edu/~chenli/pub/www2009-tastier-fuzzy.pdf. Given a keyword (which may be a partial word input), we will traverse the trie structure to look at the most probable leaf nodes given that keyword as a prefix. This assumes that the keyword is correctly spelled. To do fuzzy search on the input, we will either try to implement the incremental/no-cache or incremental/cache version detailed in the paper above. In the latter (which relies on the algorithm evaluating the input as the user types in additional letters), we will have to keep a running list of 'active nodes' in the trie that are probable prefixes given the current one, identified by edit distance and by the list of active nodes from the previous iteration. With each

In either case, the algorithm calculates active nodes by considering prefixes that are a certain edit distance away from the current prefix. In this respect, this extension is similar to the autocorrect algorithm above, except that it returns inputs that are not necessarily complete words. Given this set of possible prefixes (active nodes), we collect all the most probable leaf nodes for each one and return the most probable one. For the incremental part of this, at each point in the input, as letters are added, we use the active node list from the previous iteration to calculate the new active node set instead of calculating a new active node set from scratch, also detailed in the paper cited above.

3) We will have a third function that incorporates the above two algorithms, and based on the probability/common-ness of the two suggested outputs (the auto-corrected word and the auto-completed word), it will return the ultimate word suggestion.

Extension function:
Function that scrapes a big text file (that incorporates several books) to chart how many time a specific word occurs in this bit text file. The corpus that we will use (http://norvig.com/big.txt) contains multiple public domain books, and words from Wiktionary and British National Corpus. This function will counter how many time each word occurs, and this will be incorporated into our word data structure to determine relative probability/commonness of word.

Function that takes input text and re-weights the word's probability in the data structure. This will supplement the probabilities/likeliness of words that we've already gathered through the journal/article scraper.

Data Structure:
Trie functions signature

Outputs a trie data structure from a list of words that groups common prefixes

```
def make_trie(*words):
        return trie
```

```
/* naive implementation of tries */
def make_trie(*words):
...     root = dict()
...     for word in words:
...         current_dict = root
...         for letter in word:
...             current_dict = current_dict.setdefault(letter, {})
...         current_dict = current_dict.setdefault(_end, _end)
...     return root
```

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
```

/* Reach goal - add support for suffix matching by implementing trie as a DAWG (Directed Acyclic Word Graph)*/

```
def edge_count (trie):
def node_count (trie):
```

combines suffixes as well as prefixes for space efficiency (e.g Pity and City would require 5 nodes instead of 8 as with a trie)

```
def compress(trie):
```

Hey guys,

So Alex and I met with Styliani today, and we talked about a few things. First, we should all attempt to finish or nearly finish the OMG Cows pset this weekend so we can focus on the project (some sort of code is due next Sunday.)

Secondly, we need to write **as much pseudo code as possible** so we can more equally divide the code; we need this for our final spec, which is due Sunday.

So we're splitting up our first testable milestone into the following parts, in order of increasing difficulty (as imagined by Alex):

1. data structure itself
    a. interface at the very least
    b. Jose has basically already done this
2. parser (not computationally interesting)
    a. given a corpus, needs to spit out a dictionary w/ word probabilities in trie form
3. way to store trie in memory (not really computationally interesting)
4. algorithm itself (which is an implementation of Levenshtein in tries)
    a. this can be done recursively as in http://stevehanov.ca/blog/index.php?id=114
    b. BUT DON'T LOOK AT THE CODE (says Stella)

Please sign up for whatever you can do according to your time commitments. Shannon and I can devote, reasonably, 6 hrs to this project each over the next week. **Please reply ASAP so we can split this up fairly.**

Other general tasks:
Stella wanted us to write the parser program by Sunday, just so we have something to start with.
Everyone needs to install Python 3.31 if they haven't already.
Everybody needs to write pseudocode for what they're responsible for so we can do the final spec by Sunday.