Project 2: Autocorrect

Due: 5 April 2022

When autocorrect is turned on for a computer or for your phone, you can type imperfectly (or, indeed, just drag your finger across the relevant "keys") and it still interprets your input as valid English. There are a few layers to this, involving both estimating what you might be trying to say (the "language model") and estimating how your fingers might slip as you try to type it (the "error model"). In this project, you'll implement a limited imitation of this kind of system.

Checkpoint

The checkpoint will give you a head-start on building your language model. For the checkpoint (next Thursday), you should have a program that:

- gets filenames from the command line arguments,
- opens the corresponding files and reads words,
- strips their punctuation (see below) and makes them all lowercase,
- stores their frequencies, and
- prints out a table with the probability of each word in the input.

Don't forget: documentation is not an explicit part of the checkpoint grade, but you will need to give me enough info, somehow, that I can be convinced you've done all the above things without having to pore over your code.

The checkpoint is due at 4pm on Tuesday, 22 March. This is a comparatively lightweight checkpoint.

What counts as "punctuation" and "words"

This is a set of rules designed to balance realism with ease of implementation.

- Anything that's not alphanumeric or a hyphen or a single-quote/apostrophe is punctuation.

- Hyphens and apostrophes that fall between letters are part of the words, not punctuation. “`can't`” and “`middle-aged`” are words.
- Single-quotes that fall at the end of a word are punctuation. “`''real''`” gets stripped to “`real`” by this rule. (Note that the apostrophe in “`commodities'`” also gets stripped, unfortunately, but we're sacrificing a little realism for a lot of simplification.)
- If something that might otherwise seem like a word has no letters in it, it's not a word. “`---`”, “`**`”, and “`$1.33`” are not words by this definition.
- Some words do have a mix of letters and numbers. “`20th`” is a word. “`'90s`” gets stripped to be “`90s`” by the earlier rules.

Suggestion: use a builtin scanner or stream to read in whitespace-delimited tokens *first*. Then write a function to take each one and strip it down and return the resulting “real word”, making sure that the function is able to return something to indicate that there was no underlying word.

If there's anything in the training that isn't covered by the above rules (or is somehow both a word and not a word, or something like that), you can treat it as not a word and skip it.

Note that later in the project, you'll want to track whether a word had sentence-ending punctuation after it. This is totally ignored in the checkpoint and in all but 3 points of the rubric but if you want to start thinking now about how to preserve that info, that's fine.

I/O spec, final version

In the final version of your program, input will come from two places. The *training* will be read in from a file (whose name is provided as a command line argument). The *user input* will come from standard input (typed at the keyboard); each newline-terminated line of input will be interpreted as a sentence, with words separated by spaces.

After each line (sentence) is read in, the system should proceed, word by word, to verify what word the user *meant* to type, by providing a numbered menu of words to choose from, in order by what the system judges most likely to be the intended word (which might be what the user typed, or might not be). After the last word is verified, the full corrected sentence is printed back out and the system waits for another line (sentence) to be typed in.

The list of suggestions can be truncated at (say) twenty options or so, but the thing the user actually typed should still be on the list somewhere. The truncation should happen *after* the sorting, i.e. if you only show twenty options they should be the *best* twenty options (plus the actual user input). (Don't do the truncation until you have the math in place to rank the candidates. Make sure if you're truncating in general to have examples prepared or some way of showing that the program can generate the complete list of candidates.)

Probably, for demonstration purposes, you will want to also print some probabilities or scores next to each word, although in a "real" UI you wouldn't normally show that to the user.

We're focusing on word choice/spelling here, so if the user has typed punctuation or uppercase letters, you can strip and lowercase it just as with the training. In a "real" UI that stuff would be restored in the final output but again, you don't have to worry about that for this project.

Noisy channels

Our ultimate goal here is to build a discriminative model that chooses between a number of possible words that a user may have *meant* to type, based on what they actually *did* type (and some context), and choosing the most likely one. That is, if we see an emitted word e,

$$\arg\max_w P(\text{Intend} = w|\text{Emit} = e)$$

We'll talk about the noisy channel model in more detail in class (or you can read about it in the book or online), but for now, suffice to say that implementing the above discriminative model requires building two component models: a language model $P(\text{Intend})$ and an error model $P(\text{Emit}|\text{Intend})$. Error models (aka "noise models") encapsulate the mechanism by which errors—in our case, typoes—occur. Given that a user *intended* one word, what's the probability of emitting that word correctly, or some similar one instead? Most of the letters will usually be typed correctly, but with some (low) probability, the user might press a different key. Or an extra key. Or they might miss a key.

In a very simple error model, we might say that when a letter substitution occurs, the choice of substitute letter has uniform probability. But really, on a standard QWERTY keyboard, substituting an H for a J is waaaay more likely than substituting, say, a Q for a J. It turns out that there are quite a few ways the brain can cause typoes, but for this project we'll assume only the more mechanical slips, involving adjacent keys, are relevant. We'll work out some more detail on this error model in class, and the final exact details will be up to you.

Language models

The idea of a generative language model is to assign probabilities to "utterances" (i.e. sentences) according to the likelihood that a speaker would ever produce them if they were producing valid English at random. A complex and advanced language model might take into account various aspects of grammar and lexical categories (parts of speech) and overall sentence structure. (What is the probability we start with a noun phrase? What is the probability that the noun phrase starts with a determinative? What is the probability that the determinative is "The"? ...etc.) In this project, though, we'll be dealing purely with word-based Markov models: they're a good tradeoff between predictive power and implementation complexity.

The simplest of the word-based language models is the *unigram* model—meaning "one word"—and the generative aspect of this is that you could imagine a babbler rolling dice before every word, and choosing the word to utter based on a weighted probability distribution of all English words. No sentence context or any other conditioning environment applies. If the word "the" occurs 4% of the time, then it will have a 4% chance of being produced (even if it was just produced, which means you have a 0.16% chance of producing the pair "the the"!). This kind of model is also known as an order-0 Markov model.

The next level of this that we'll see is a *bigram* ("two word") model. Here, the probability of producing a word is conditioned on what word came before it (or, if it is the first word of an utterance, conditioned on that fact). The unigram probability of a word like "is" is relatively high, occurring a bit less than 1% of the time; but the probability of "is" given that the previous word was "the", that is,

$$P(w_i = \text{is} \,|\, w_{i-1} = \text{the})$$

, is vanishingly small. The phrase "the is" just doesn't occur.[1] A bigram model is an example of a first-order Markov model.

In practice, the bigram model is great when there's enough data—that is, when the previous word is a frequent one—but with less common words it has a harder time. There are various ways to mitigate this but the easiest is to interpolate, or take a weighted average, of the unigram and bigram probabilities. For our purposes, we'll make the weighting constant: give 0.8 weight to the bigram model and 0.2 to the unigram.

Note that the "preceding word" here should be handled sequentially in the final program, letting the user select their intent for one word before computing values for the next. For instance, if the user typed "`the opwn fokder`", and ends up correcting "opwn" to "open", then "open" should be used as the preceding word when evaluating probabilities for replacements for "fokder".

Unknown words handling

Finally, there is a question of what to do with words that are not just rare but entirely unknown in the training. In some applications we can glibly assign a very low "probability" to such cases, but this breaks the generative distribution, and anyway there are more elegant solutions. In our language model, we can observe that words that appear only once in the entire training set are appearing in the same sorts of environments that unknown words would show up in, and use this information to manufacture statistics for a special pseudo-word "UNK". In the unigram model, count the number of words that only appear once in the training,[2] and then add the word UNK to the model with a frequency matching that count. Similarly, add UNK to the counts for the bigram model, both as a "current" word and as a "preceding" word.

The first, easier way to use those counts is in the bigram model. When the preceding word is unknown to the model, you can just look up the probabilities where UNK is the previous word, and use those.

[1]Except when, rarely, it does—as in fact it does on this page! This is why I so dislike assigning zero probabilities.

[2]Trivia: such a word is called a *hapax legomenon*, plural *hapax legomena*.

The other is when the current word under consideration is unknown—ie the user has typed something and the model is ranking its likelihood. It could mean a typo, or it could mean it's just a legitimate new word. But to handle this probability generatively you need to know not only the probability of generating *an* unknown word, but the probability of *this particular* unknown word: this length, these particular letters, and so on. Jointly, that's something like

p(UNK, length, each letter)

which we can chain into

p(UNK) * p(length | UNK) * p(each of the letters | UNK, length)

Which might seem complicated, but a few independence assumptions and insights make it straightforward. The first term is just the probability of the pseudo-word UNK (described earlier). If known and unknown words have the same length distributions (let's assume yes), then the second term is just p(length of this word), a statistic that is easy to track when building the model. And finally, if each letter were equiprobable and there's *length* letters, then the third term is just

$$\left(\frac{1}{28}\right)^{\text{(length of the UNK word)}}$$

So processing unknown words involves several moving parts, but is relatively straightforward. (Why 28? Because we're treating apostrophe and hyphen as pseudo-letters, plus the 26 actual letters of English.)

Training data

You should build your own training data with a very small number of words in it, in order to test your code with known data. But your program should also be able to work with real data, at scale. To that end, in the directory `/home/shared/nlp` there are files `nanc20.txt` through `nanc2000.txt` that are increasingly larger subsets of the contents of the North American News Corpus. Don't copy them—they are licensed—but your program should be able to make use of them for training.

It's ok if your program takes a few minutes to read in and do the initial processing of the largest of them, but a) it still shouldn't take (say) *hours*, and b) the actual processing of the standard input, with suggested spelling corrections and so on, should still be pretty snappy. This has some *very* important implications on your data structure choices.

NB: These data files are encoded in the iso-latin-1 character set, which will trip up the default input scanner in Java or Python. If you're not using C++, make sure to set the input encoding accordingly.

Final version

A full-credit final version will be a complete, non-buggy, working implementation of a noisy-channel typo correction system, TOGETHER WITH convincing proof that it is correct. The "proof" should consist of test cases (in whatever format is convenient to you) to illustrate various situations, including both input and expected results.

Remember that there need to be clear instructions on how to run it in general as well as how to run each/all of the tests and quickly verify that they ran correctly (and which rubric items each one corresponds to); and don't forget to explain how to enter actions and interpret the display! Having complete and correct documentation is an easy 15 points, but if your documentation omits important info or tells me the wrong thing, you'll get less than full credit there.

After checkpoint work (15 points) and documentation (15 points), there remain 70 points in the rubric, which will be awarded according to the table below. Note that number of points does *not* necessarily correspond to difficulty; and you should probably implement the first items of each rubric group before you move on to any other part of the implementation. (*Within* groups they generally proceed in order of suggested implementation.)

Score	Description
	Reading and correcting the user input
4	Reads whole lines of standard input as distinct utterances, breaks them into words (lowercase, stripped punctuation). Prints back full sentence in this form (with words as edited, if menu of candidate words is implemented).
4	The program can compute noisy channel model scores for individual words.

For each word of input, program generates and prints a complete list of candidates...

3 ...of same length, with exactly one letter changed.

3 ...with one inserted letter.

3 ...with one deleted letter.

5 ...with insertions, changes, or deletions, up to an edit distance of 2.

Candidate words menu:

4 Each word of input's candidate lists (plus the original word as typed) is printed in a numbered menu for user to select from, and resulting "corrected" sentence is printed after all words selected.

4 Candidate lists are sorted by language-model-based likelihood score, which is printed next to each candidate (including the original word). (Score should be based on full noisy-channel model if error model is implemented).

Error model

Error model assigns probabilities to each candidate in list (and, in regular or in debugging output, prints them) according to...

4 † ...fixed probability .973 per letter of no change and .001 per letter of changing to each of 27 other possible letters (incl hyphen, apostrophe).

4 † ...fixed probability model allowing for insertions and deletions but each possible letter has equal probability in generative model.

8 † ...probability model permitting only fixed-length typoes (no insertion or deletion) but accounting for keyboard adjacency when generating typoes. (If you have insertion/deletion working in your "generating possible other words" part of the program, but haven't worked out how to score them, you can print them out with their language model scores multiplied by typo model scores of "???" or 0 or whatever, in order to claim the points from those other categories even if they're not fully worked into the typo model.)

10 † ...probability model permitting all three edit types that accounts for keyboard adjacency when generating changes and insertions.

† The three error-model rubric items are mutually exclusive, i.e. you get 4 *or* 8 *or* 10 points for these, not a total of up to 18.

Language model

Training language model:

10	Reads training data from files, strips punctuation, and prints (in either regular or debugging output) unigram probabilities per the checkpoint description. **(This is the checkpoint.)** Does so without taking too long, and subsequent accesses (to generate the candidate lists) are very fast, even on very large training data sets.
3	In the training, sentence-ending punctuation is treated as divider between utterances (and if bigrams are used in the language model, this information is incorporated).

Language model evaluates (and, in regular or debugging output, prints)...

3	...unigram probability of each word of input.
6	...bigram probability of each word of input (with "previous word" as edited, if menu of candidate words is implemented).
3	...linear interpolation of unigram and bigram probability of each word of input.
5	...bigram or interpolation, with appropriate unknown-word handling.

Handing in

The checkpoint and the final version are due at 4pm on their respective due dates. Hand them in as `proj2` using the handin script.