Project 2: Swype

Due: 5 April 2016

Swype is an input system for phones that lets you type imperfectly (or, indeed, just drag your finger across the relevant "keys") and 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 system.

Checkpoint

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

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

When stripping punctuation, for now you need only look at the start and end of a token; remember that hyphens and apostrophes are generally considered part of the word they're inside!

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!

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,

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

, 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.

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

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

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 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. For the bigram model, you can similarly aggregate the counts of words that occur immediately after unique words, store them as occurring after UNK as well, and use those counts to estimate bigram probabilities of words given that the previous word was unknown.

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

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

out some more detail on this error model in class, and the final exact details will be up to you.

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.

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 25 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 (25 points) and documentation (25 points), there remain 100 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.)

If your program is too slow to work effectively on real data, it won't get full credit. I am still working out how that will affect the rubric. If you need to know more about how it will affect the rubric, come talk to me.

Score	Description
	Input handling:
10	Reads training data from files, strips punctuation, and prints (in either regular or debugging output) unigram probabilities per the checkpoint description.
5	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), and punctuation-only words (which turn into empty strings) are removed gracefully.
5	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).
	Language model evaluates (and, in regular or debugging output, prints)...
5	...unigram probability of each word of input.
10	...bigram probability of each word of input (with "previous word" as edited, if menu of candidate words is implemented).
5	...linear interpolation of unigram and bigram probability of each word of input.
10	...bigram or interpolation, with appropriate unknown-word handling.
	For each word of input, program generates and prints a complete list of candidates...
5	...of same length, with exactly one letter changed.
10	...with one inserted letter.
5	...with one deleted letter.
5	...with insertions, changes, or deletions, up to an edit distance of 2.

Candidate words menu:

5 Each word of input's candidate lists (plus the original word as typed) is printed in a numbered menu for user to select from...

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

5 † ...fixed probability .975 per letter of no change and .001 per letter of changing to each of 25 other possible letters.

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

15 † ...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 5 *or* 15 points for these, not a total of up to 25.

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.