Building a spell checker

PURPOSE

Build summary tables from a real-world dataset that will be used to implement Kernighan's spelling algorithm.

Use defaultdict, exception handling, regular expressions, list comprehensions, objects and dictionaries in a real-world size project.

Learn useful design techniques including the use of an intermediate file and the ability to sort data structures and objects via the use of a constructed sort key.

Learn useful pieces of Python infrastructure including the use of the time class and the use of json for creating persistent data.

SPECIFICATIONS FOR DATA CLEANUP PROGRAM

1. Use a command line parameter to provide the name of the test file. If no parameter is used, the default should be the full AP corpus,

home/turing/t90rkf1/d503/dhw/data/ap88.txt

Use exception handling to prevent your program from crashing if the user enters an incorrect filename.

Do the following for each record in the input file. Don't try to read the whole file at once, and don't use append; both will make your program too slow to finish in the 8 CPU minutes allotted per process on turing/hopper.

2. Clean up the data.

a) Pick off the AP tag using a regular expression. You could also use split().

b) Replace every character that is not an Ascii letter by a space.

That will split abbreviations into two words, e.g., 'doesn' + 't'. That is a common approach in NLP software, e.g., in the Stanford parser.

On the other hand, that will also drop all of the punctuation. Even if you don't need the punctuation, that is generally not a good idea because it doesn't allow you to track back to the sentence where a word was found. However, it is convenient and simple, so we will do that here.

c) Per convention, convert to lowercase.

d) Use split() to break up the words.

3. Use defaultdict to create dictionaries of character unigrams (single characters) and character bigrams (pairs of characters) with their frequencies, including beginning-of-word and end-of-word as shown in the sample output.

One way to do this is to use a regular expression (check out the ‘w’ format) to surround the tokens by angle brackets, e.g., '<word>'. That produces the following unigrams, where '<' means beginning of word and '>' means end of word:

'<', 'w', 'o', 'r', 'd', '>'.

It also produces the following bigrams:

'<w', 'wo', 'or', 'rd', 'd>'.

Another approach would be to process each word using a loop, adding to the appropriate unigram and bigram buckets. Then add to appropriate buckets for beginning of word and end of word.

Initialize the dictionaries before reading the file, then update them after every record is read.

4. Build a dictionary of words in the database with their frequencies. Do this in parallel with the previous item, i.e., you only need to read each record once, then process it several ways.

As a check on the file processing, print the following items at the end of the program:
- the number of distinct unigrams
- the number of distinct bigrams
- the number of words in the file
- the number of items in the dictionary
- the ten most frequent words along with the number of times each occurs

The latter can be accomplished with options of sorted(); you can build the key with a lambda function or with operator.itemgetter().

5. Since some of these are time-consuming operations, measure the elapsed time and CPU time to process the file. Use time.perf_counter() to measure elapsed time and time.process_time() to measure CPU time. Call each function before and after reading the file and take the difference. Print the results in the indicated format.

6. Save all the data structures in a dictionary so that it can be serialized. Use json to serialize the dictionary and create a persistent data structure that is human as well as machine readable. The goal is to convert it to a format that can be written to a file and read back in and reconstructed. We don’t want the actual spelling correction program to recreate this preparatory work.

To give the file a canonical form so that they can be compared, sort the three dictionaries first and use the keys unigrams, bigrams and words. In other words, your dictionary should look like this:

<name> = {'unigrams': <name>, 'bigrams': <name>, 'words': <name>}

Call your file json-data.json.

SPECIFICATIONS FOR THE SPELL CHECKER

1. Assume the misspelled words are provided as command line parameters. Quit with a message if there are none provided.

2. For each word, make sure that it doesn't contain any non-alphabetic characters. If it does, reject it with a message. Then create a list of possible corrections as suggested by Kernighan et al. The possible corrections include:

a) Add any letter at any position (including at the beginning or end of a word).

b) Delete the letter at any position.

c) Substitute the letter at any position with any other letter.

d) Transpose any two consecutive letters.

Consider these corrections independently, i.e., we will only evaluate the effect of one error on a word at a time. Use the formulas provided in the slides.

Prune the list by looking up each possible correction in the dictionary. Drop any possible correction that is not in the dictionary, i.e., we will ignore any string which is not a word or is so rare that it does not appear in the corpus at all.

For each of the four categories, print the number of possible corrections and the number that are left after pruning using the layout provided in the sample output.

A link to the original paper can be found on the course home page:

> Kernighan, M., Church, K., and Gale, W. (2000). A spelling correction program based on a noisy channel model. In COLING '90 (Helsinki), v. 2, pp. 205–211.

3. For each of the misspelled words, use Kernighan's approach to find the probability of each possible correction.

Define a Correction class containing the fields you will need to print the following table:

Candidate Correction	Error Type	Error Pos	Correct Letter	Error Letter	x|w	P(x|word)	P(word)	10^9 * P(x|w)P(w)
actress	del	2	t	c	c|ct	0.000116531	0.000058411	6.806804
across	subst	3	o	e	e|o	0.000007332	0.000245289	1.798504
acres	ins	4	-	s	es|e	0.000019887	0.000081894	1.628671
acres	ins	5	-	s	ss|s	0.000017237	0.000081894	1.411670
acrss	ins	2	-	e	re|r	0.000007927	0.000000042	0.000333
adress	subst	1	d	c	c|d	0.000002198	0.000000098	0.000216
agress	subst	1	g	c	c|g	0.000001429	0.000000070	0.000100
caress	trans	0	ca	ac	ac|ca	0.000000000	0.000000126	0.000000
access	subst	2	c	r	r|c	0.000000000	0.000066566	0.000000

You might wonder why three non-words, *acrss*, *adress* and *agress*, are included. That's because they are included in the AP88 corpus, so the system thinks they are words. Cleaning up the corpus would require another source of correctly spelled words, so that is a separate project.

Create an object for each possible correction containing the data in the table. Save the items in a list, then sort them so that the chart prints in descending order of probability.

To do this you will need the confusion matrices from Kernighan's paper. They are available on turing at ~t90rkf1/d503/dhw/hw04-spell/tables.py. Use the following code to import them into your program:

```
import sys
sys.path.insert(0, '/home/turing/t90rkf1/d503/dhw/hw04-spell/')
import tables
```

This code temporarily adds the directory for this assignment to the top of the list that Python uses to search for libraries.

To index into these tables, you need to know the position of a given letter in the alphabet. You can use the find() function of the string class and the constant string.ascii_lowercase to obtain this information. Do not use ord()– that is not a Pythonic approach. Another option is to write a function that converts the format given to a dict of dict which you can then index directly. You can do that in a few lines of code using a two-level dict comprehension.

The table above differs from what a real spell checker would print because it counts each case of “acres” differently. To combine them, we would have to do the following:

- Sort the objects by proposed word so we can see the duplicates.
- Add the probabilities for duplicate entries so we can get a realistic probability for the word.
- Resort the entries by combined probability so that the words will be listed in order.
- Print a reduced table containing only the word and its total probability, as the other fields make no sense once the probabilities are combined.

All tables must have aligned columns. Integers must be right-aligned. Real numbers must be decimal-aligned.

Every number you print must have a rubric, e.g., as a row or column heading.

Use list comprehensions wherever possible.

ABOUT THE CORPUS:

The AP ’88 corpus is a compilation of news articles published by the Associated Press in 1988. The Associated Press is a consortium of newspapers. Members upload articles written by their journalists and can download and publish articles written by journalists working for other members. That enables newspapers to print articles to print articles about news events happening in all parts of the world even though it is not practical to have reporters everywhere at all times.

The AP has a website at https://apnews.com/, and its competitor UPI (United Press International) has a website at https://www.upi.com/.

The file has been extracted from TIPSTER v. 2, available from the LDC, the Linguistic Data Consortium hosted at the U. of Pennsylvania. *Please note that the corpus has been licensed only for the purpose of this class. You may not keep a copy or use it for other purposes after the class is over*. You can keep and reuse the output of this program, which is a series of summary tables, so that you will be able to run the spell checker (hw04b.py) after the class is over.

TIPSTER v. 2 is also called TREC-2, or the Text Research Collection v. 2. You can learn more about the corpus here:

> https://catalog.ldc.upenn.edu/LDC93T3C
> https://catalog.ldc.upenn.edu/docs/LDC93T3C/tipster.readme.html

The TIPSTER project was sponsored by the Software and Intelligent Systems Technology Office of the Advanced Research Projects Agency (ARPA/SISTO) in an effort to advance the state of the art in information retrieval and data extraction from large, real-world data collections.

The full TIPSTER data includes a test collection built at NIST for both TIPSTER and TREC, which is an annual set of contests on informational retrieval. NIST is the National Agency of Standards and Technology, an agency of the U.S. government. You can learn more about TREC at https://trec.nist.gov/. TREC contests contain queries about news articles and other documents. The contestants, who are research teams from different universities, attempt to provide either the most relevant documents from a given set, or actual answers to the questions.

Both as part of the philosophy of leaving the data as close to the original as possible, and because it is impossible to check all the data manually, there are many errors in the data. These include errors in the original data, such as problems in the originally uploaded new stories and other typographical errors, as well as errors that may have been introduced in the automated reformatting done at the University of Pennsylvania and at NIST.

The error checking concentrated on readability of the data rather than on content. This included checking for control characters and syntactic items such as correct matching of begin and end tags and document numbers in order to make the files readable. They did not attempt to correct sentence fragments, formatting around tables, misspellings, missing fields and related issues.