Problem Set #7

Fall 2022

This assignment is purely a programming one. Instead of the usual submission requirements, submit one `zip` file containing all four Python files required in the assignment. The file should be named `hw_id_07.zip` with `id` replaced by your ID as usual. You may also use `rar` or any other compression format you prefer.

1 General outline

Subject(s): language models, n-grams, simulated annealing

A North Korean spy, imprisoned in a local prison in Los Angeles, was caught sending an encrypted message written using an encryption system known as the Substitution Cipher. That is, each alphabetic letter is replaced with another letter. Scientists from a local university were the ones to intercept the message (provided on Piazza as `problemset_07_encrypted_input.txt`), and they ask for your help: find the encryption key (i.e. letter permutation) used by the spy! Decrypt the message! Save the world! In order to solve this problem, we will build four Python modules (i.e., files), each serving a different purpose as explained below:

1. **A language model module**, containing the functionality needed in order to read a corpus from an online resource and build an n-gram language model (in particular, a bigram language model) using that corpus. The module name should be: `language_model.py`.
2. **A permutation module**, representing a letter permutation, according to the description below. The module name should be: `permutation.py`.
3. **A simulated annealing module**, containing an implementation of the simulated annealing algorithm for finding the optimal hypothesis in a large conjecture space, as discussed in class the module name should be: `simulated_annealing.py`.
4. **A main module**, connecting all the logical parts that have been described so far (use import statements in order to import relevant module functionality from previously written modules). The module name should be: `main.py`.

The next sections provide a detailed description of each module.

2 Python modules

2.1 language_model.py

This module must contain two classes: `CorpusReader` and `LanguageModel`, in addition to any other functionality as you see fit. Classes' description is as follows:

1. The `CorpusReader` class
   This class should contain all relevant data structures and methods needed to read an online corpus from a given URL, and prepare it for the statistical counts.
   It should have a constructor which receives a URL indicating the location of the online corpus, and:
   - Opens the file, reads its contents and closes it (releases resources). Use Python's urllib module.
   - Cleans the read corpus up (i.e. filters it) so that it only contains characters from the English alphabet (a-z), spaces, and characters from the following set which we shall call T:

,	.	:	\n	#	(	)	!	?	\'	\"

     Filtering should also turn the entire corpus to lowercase letters.

2. The `LanguageModel` class
   This class should contain all relevant data structures and methods needed to perform statistical counts (unigrams, bigrams, MLE etc), given a filtered corpus.
   It should have a constructor which receives a filtered corpus (which is provided as an initialized instance of the `CorpusReader` class), and:
   - Uses the given corpus to gather unigram and bigram raw counts (for characters), as was discussed in class. So basically, you need to find: $\forall w_i \in \Sigma$, $count(wi)$ and $\forall (wjwi) \in \Sigma$, $count(w_j, w_i)$ as a preliminary step towards finding the unigram and bigram probabilities.
     You may assume that the alphabet is composed of the English alphabet characters (lowercase) + the set of characters described above.
     Basically, $\Sigma = \{a, b, c, ..., z\} \cup \{space\} \cup T$, with space being the regular space character, and T being the set of characters in the table above.
   - Uses the above counts to obtain MLE unigram and bigram probabilities and apply Laplace (add 1) smoothing, as was discussed in class. These probabilities should be saved as class instance variables for future use. This will be your language model.

2.2 permutation.py

This module must contain one class named `Permutation` that represents a permutation (recall that a permutation is simply a bijective function from Σ to itself), in addition to any other functionality as you see fit.
It should have a constructor which receives some mapping between the characters (the way to implement this mapping is up to you). In addition, the class should have the following methods:

1. `get_neighbor` - returns a random neighbor of the current permutation instance. A neighbor of a permutation is defined as a new permutation that simply replaces the mapping of two randomly

chosen characters from Σ. Use the `choice` method from the `random` module which takes a list as an argument, and returns a randomly selected element from that list.

2. `translate` - receives an input string, and returns the translation of that string according to the current permutation instance (i.e. takes each character of the string and replaces it with its corresponding character according to the permutation).

3. `get_energy` - receives a data argument (= encrypted message) and a language model, and returns the energy of the current permutation.
   Note: the energy of the permutation is the result of translating the encrypted message according to the permutation key, and evaluating the probability of this translated sequence of characters according to the language model. That is, if the translated sequence of characters is $w_1, ..., w_n$ then we look for $P(w_1, w_2, w_3, ..., w_n)$. We saw that according to a bigram language model, this value is approximated as: $P(w_1) \cdot P(w_2|w_1) \cdot ... \cdot P(w_n|w_{n-1})$. We also saw that since the product of many such probabilities can be a very small number (and may thus be automatically rounded to 0 by the Python interpreter), we can use the logarithmic function and apply it on the product:
   $log_2(P(w_1) \cdot P(w_2|w_1) \cdot ... \cdot P(w_n|w_{n-1})) = log_2((P(w_1)) + log_2(P(w_2|w_1)) + ... + log_2(P(w_n|w_{n-1}))$
   In addition, recall that the pseudocode for simulated annealing finds a minimum point of the energy function, while we want to find a maximum point. Therefore, we should take the negative of the whole expression:
   $-log_2(P(w_1) \cdot P(w_2|w_1) \cdot ... \cdot P(w_n|w_{n-1})) = -(log_2((P(w_1)) + log_2(P(w_2|w_1)) + ... + log_2(P(w_n|w_{n-1}))) =$
   $-log_2((P(w_1)) - log_2(P(w_2|w_1)) - ... - log_2(P(w_n|w_{n-1})))$
   To sum up, the energy of the permutation should be: $-log_2(P(w_1)) - log_2(P(w_2|w_1)) - ... - log_2(P(w_n|w_{n-1}))$ Use the `log` function from the `math` module which takes a value v and a base b as arguments, and returns $log_b v$. Use base 2 for the log.

2.3 simulated_annealing.py

This module must contain one class named `SimulatedAnnealing` that implements the simulated annealing algorithm as discussed in class.
It should have a constructor which receives an initial temperature, a threshold and a cooling rate (in that order). In addition, the class should have the following method:

- `run` - receives an initial hypothesis (= initial permutation), an encrypted message and a language model, and returns the output of the simulated annealing loop, as provided in the psuedocode in the simulated annealing handout. Use the `exp` function from the `math` module, which takes a value v and returns e^v.
  In order to decide whether or not to switch to the neighbor, use the random method from the `random` module which returns a random real number $r \in [0, 1]$. Take r and the probability of the switch, p, and check whether $r < p$. If so, we switch to our neighbor, otherwise we stay with the current hypothesis.

2.4 main.py

This module must contain a main function, which incorporates all previously written modules. It should:

1. Read the online corpus http://www.gutenberg.org/files/76/76-0.txt (from project Gutenberg).
2. Create a bigram language model.
3. Read the encrypted message from `problemset_07_encrypted_input.txt` on Piazza.
4. Create an initial permutation hypothesis. Try starting with the most simple hypothesis, which maps each letter in Σ to itself.
5. Run the simulated annealing algorithm with the relevant parameters.
   Try using an initial temperature of 10, 100 or 1000, a cooling schedule of 0.95, 0.995 or 0.9995, and a threshold of $10^{-1}, 10^{-3}$ or 10^{-5}. This is just a suggestion - you are encouraged to use other values and other combinations as well.
6. Print out:
   - The winning permutation.
   - The initial temperature, threshold and cooling rate used.
   - The content of the deciphered message.