# Project 5 (Markov Model)

## Clarifications and Hints

## Prologue

Project goal: use a Markov chain to create a statistical model of a piece of English text and use the model to generate stylized pseudo-random text and decode noisy messages

The zip file (`http://www.swamiiyer.net/cs110/markov_model.zip`) for the project contains

- project specification (`markov_model.pdf`)
- starter files
  - `markov_model.py`
  - `text_generator.py`
  - `fix_corrupted.py`
- test script (`run_tests.py`)
- test data (`data/`)
- report template (`report.txt`)

This checklist will help only if you have read the writeup for the project and have a good understanding of the problems involved. So, please read the project writeup before you continue with this checklist.

## Prologue

Understanding how dictionaries work is crucial for this project, so make sure you understand the following example which illustrates how you create a dictionary of dictionaries and how you manipulate them

```
>>> M = {}                                # create an empty dictionary M
>>> M.setdefault('ba', {})                # add key/value pair 'ba'/{} to M
{}                                        # since 'ba' didn't exist in M,
                                          # {} (the value just added) is returned
>>> M                                     # check M
{'ba': {}}
>>> M['ba'].setdefault('n', 0)            # add key/value pair 'n'/0 to the
                                          # dictionary M['ba']
0                                         # since 'n' didn't exist in M['ba'],
                                          # 0 (the value just added) is returned
>>> M                                     # check M
{'ba': {'n': 0}}
>>> M['ba']['n'] += 1                     # increment the value corresponding to the
                                          # key 'n' in the dictionary M['ba'] by 1
>>> M                                     # check M
{'ba': {'n': 1}}
>>> M['ba'].setdefault('n', 42)           # add key/value pair 'n'/42 to the
                                          # dictionary M['ba']
1                                         # since 'n' exists in M['ba'],
                                          # setdefault() simply returns (without
                                          # changing) the corresponding value, 1
>>> M                                     # check M
{'ba': {'n': 1}}
>>> M.setdefault('an', {})                # add key/value pair 'an'/{} to M
{}
>>> M                                     # check M
{'ba': {'n': 1}, 'an': {}}
```

```
>>> M['an'].setdefault('a', 0)      # add key/value pair 'a'/0 to the
                                    # dictionary M['an']
0
>>> M                               # check M
{'ba': {'n': 1}, 'an': {'a': 0}}
>>> M['an']['a'] += 1               # increment the value corresponding to the
                                    # key 'a' in the dictionary M['an'] by 1
>>> M                               # check M
{'ba': {'n': 1}, 'an': {'a': 1}}
>>> M['an']['a'] += 1               # increment the value corresponding to the
                                    # key 'a' in the dictionary M['an'] by 1
>>> M                               # check M
{'ba': {'n': 1}, 'an': {'a': 2}}
>>> M.keys()                        # get the keys of M
['ba', 'an']
>>> M.values()                      # get the values of M
[{'n': 1}, {'a': 2}]
>>> M['ba'].keys()                  # get the keys of M['ba']
['n']
>>> M['ba'].values()                # get the values of M['ba']
[1]
>>> M['an'].keys()                  # get the keys of M['an']
['a']
>>> M['an'].values()                # get the values of M['an']
[2]
```

## Problems

Problem 1 (*Markov Model Data Type*) Create a data type `MarkovModel` to represent a Markov model of order $k$ from a given text string, and supporting the following API:

| method | description |
|---|---|
| `MarkovModel(text, k)` | create a Markov model `model` of order $k$ from $text$ |
| `model.order()` | order $k$ of Markov model |
| `model.kgram_freq(kgram)` | number of occurrences of $kgram$ in text |
| `model.char_freq(kgram, c)` | number of times that character $c$ follows $kgram$ |
| `model.rand(kgram)` | a random character following the given $kgram$ |
| `model.gen(kgram, T)` | a string of length $T$ characters generated by simulating a trajectory through the corresponding Markov chain, the first $k$ characters of which is $kgram$ |

Hints

- Instance variables
  - Order of the Markov model, `_k`
  - A dictionary to keep track of character frequencies, `_st`

## Problems

- `MarkovModel(text, k)`
  - Initialize instance variables appropriately
  - Construct circular text `circ_text` from `text` by appending the first `k` characters to the end; for example, if `text = 'gagggagaggcgagaaa'` and `k = 2`, then `circ_text = 'gagggagaggcgagaaaga'`
  - For each `kgram` from `circ_text`, and the character `next_char` that immediately follows `kgram`, increment the frequency of `next_char` in the dictionary `_st[kgram]` by 1; for the above example, the dictionary `_st`, at the end of this step, should look like the following:

```
{'aa': {'a': 1, 'g': 1},
 'ag': {'a': 3, 'g': 2},
 'cg': {'a': 1},
 'ga': {'a': 1, 'g': 4},
 'gc': {'g': 1},
 'gg': {'a': 1, 'c': 1, 'g': 1}}
```

- `model.order()`
  - Return the order of the Markov model
- `model.kgram_freq(kgram)`
  - Return the frequency of `kgram`, which is simply the sum of the values of `_st[kgram]`
- `model.char_freq(kgram, c)`
  - Return the number of times `c` immediately follows `kgram`, which is simply the value of `c` in `_st[kgram]`

## Problems

- `model.rand(kgram)`
    - Use `stdrandom.discrete()` to randomly select and return a character that immediately follows `kgram`
- `model.gen(kgram, T)`
    - Initialize a variable `text` to `kgram`
    - Perform `T - _k` iterations, where each iteration involves appending to `text` a random character obtained using a call to `self.rand()` and updating `kgram` to the last `_k` characters of `kgram`
    - Return `text`

## Problems

Problem 2 (*Random Text Generator*) Write a client program `text_generator.py` that takes two command-line integers $k$ and $T$, reads the input text from standard input and builds a Markov model of order $k$ from the input text; then, starting with the $k$-gram consisting of the first $k$ characters of the input text, prints out $T$ characters generated by simulating a trajectory through the corresponding Markov chain, followed by a new line.

Hints

- Read command-line arguments `k` and `T`
- Initialize `text` to text read from standard input using `sys.stdin.read()`
- Create a Markov model `model` using `text` and `k`
- Use `model.gen()` to generate a random text of length `T` and starting with the first `k` characters of `text`
- Write the random text to standard output

## Problems

Problem 3 (*Noisy Message Decoder*) Write a client program `fix_corrupted.py` that takes an integer $k$ (model order) and a string $s$ (noisy message) as command-line arguments, reads the input text from standard input, and prints out the most likely original string.

Hints

- Main idea behind `model.replace_unknown(corrupted)`

  When we fix the corrupted messages, we have to look at the missing letter in the context of what comes before it and what comes after it. For example, let the corrupted text be `'it w~s th'`, `k = 4`, and let the characters that follow the 4-gram `'it w'` be `'a'`, `'b'`, and `'c'`. So you want to pick the best of three hypotheses (call them $H_a$, $H_b$, and $H_c$). Let's use the notation `'abcd'|'e'` to mean the probability of finding an `'e'` after the 4-gram `'abcd'`. This probability is 0 if `'e'` does not follow `'abcd'` in the text.

  The likelihood of $H_a$ is the product of $(k+1)$ probabilities: `'it w'|'a'`, `'t wa'|'s'`, `' was'|' '`, `'was '|'t'`, and `'as t'|'h'`.

  The likelihood of $H_b$ is the product of the following $(k+1)$ probabilities: `'it w'|'b'`, `'t wb'|'s'`, `' wbs'|' '`, `'wbs '|'t'`, and `'bs t'|'h'`.

  The likelihood of $H_c$ is the product of the following $(k+1)$ probabilities: `'it w'|'c'`, `'t wc'|'s'`, `' wcs'|' '`, `'wcs '|'t'`, and `'cs t'|'h'`.

  Now, the character that you use to replace `~` with is the one with the maximum likelihood. So if $\max(H_a, H_b, H_c) = H_a$, then you would replace `~` by the character `'a'`. Use the `argmax()` function for this.

## Problems

- Pseudocode for `model.replace_unknown(corrupted)`

```
if corrupted[i] == '~':
    kgram_before = kgram before ~
    kgram_after = kgram after ~
    probs = []
    for each hypothesis from hypotheses (characters that can replace ~):
        context = kgram_before + hypothesis + kgram_after
        p = 1.0
        for i from 0 to _k + 1:
            kgram = kgram from context starting at i
            char = character from context that follows kgram
            if kgram or char is non-existent, then set p to 0 and break
            Otherwise, multiply p by probability of char following kgram
        add p to probs
    add to original the hypothesis that maximizes probs (use argmax())
```

- Implement `fix_corrupted.py` as follows:
  - Read command-line arguments `k` and `s`
  - Initialize `text` to text read from standard input using `sys.stdin.read()`
  - Create a Markov model `model` using `text` and `k`
  - Use `model.replace_unknown()` to decode the corrupted text `s`
  - Write the decoded text to standard output

## Epilogue

Your project report (use the given template, `report.txt`) must include

- time (in hours) spent on the project
- short description of how you approached each problem, issues you encountered, and how you resolved those issues
- acknowledgement of any help you received
- other comments (what you learned from the project, whether or not you enjoyed working on it, etc.)

Before you submit your files

- make sure your programs meet the input and output specifications by running the following command on the terminal

```
$ python run_tests.py -v [<problems>]
```

- make sure your programs meet the style requirements by running the following command on the terminal

```
$ pep8 <program>
```

- make sure your report isn't too verbose, doesn't contain lines that exceed 80 characters, and doesn't contain spelling/grammatical mistakes