Part I: Regex and Finite State Machines (Using Perl Regular Expression)

Q1. [10 points] Write regular expressions for the following languages:
a) The set of all strings with two consecutive repeated words (e.g. “Humbert Humbert” and “the the”)

A: “([a-zA-Z]+\s+\1)”

b) The set of all strings from the alphabet a,b such that each a is immediately preceded **or** immediately followed by a b
A: ”(ab|ba)”

c) Dates in the form mm/dd/yy. Make sure months and days are in the valid range (1-12 for months, 1-31 for days), but you don’t need to specify agreement (for example, 02/31/98 is still fine).
A: "(0[1-9]|1[0-2])(/)(0[1-9]|[12][0-9]|3[0-1])(/)"

(Ref: https://www.regular-expressions.info/dates.html)

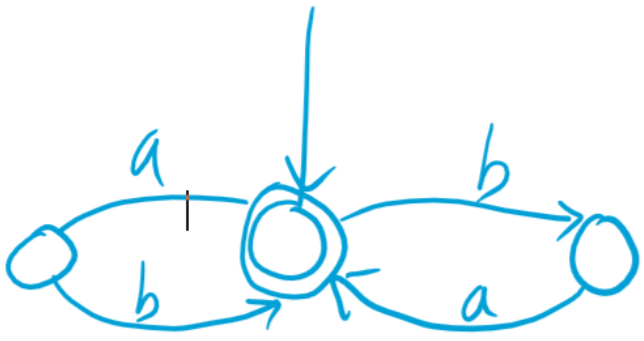
d) All strings that start at the beginning of a line with an integer, and end at the end of the line with a word.
A: “(^\d+)([a-zA-Z]+$)”

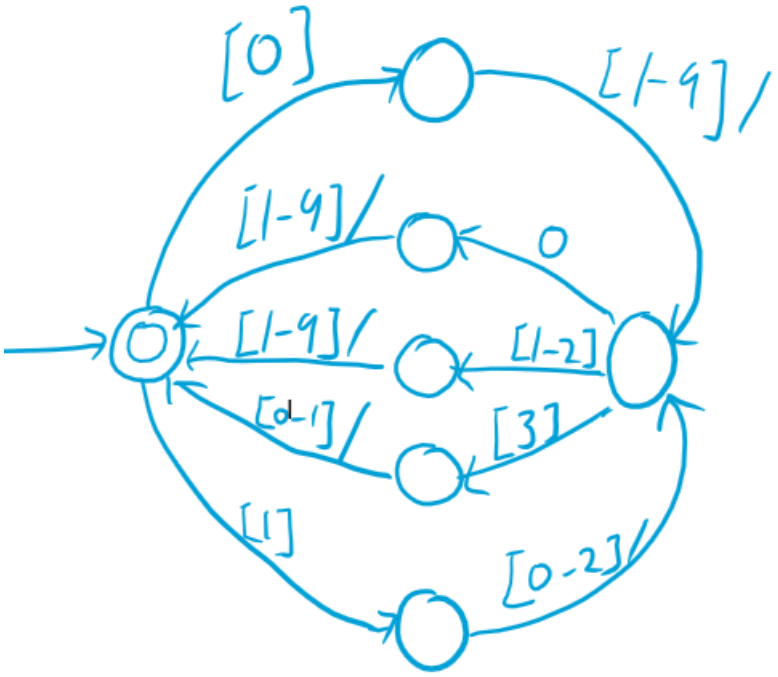
e) All inflections of the verbs “walk”, “talk”, and “break”.
“Note: the inflectional forms for break --> break breaks broke broken breaking”

A: "(\bbreaks?)|(\bbroke(n?)|(ing))|(\bwalk(s?|(ed)|(ing)))|(\btalk(s?)|(ed)|(ing))"

Q2. [5 points] Draw a Finite State Automata (FSA) that corresponds to the languages in parts (b) and (c) above.

b.





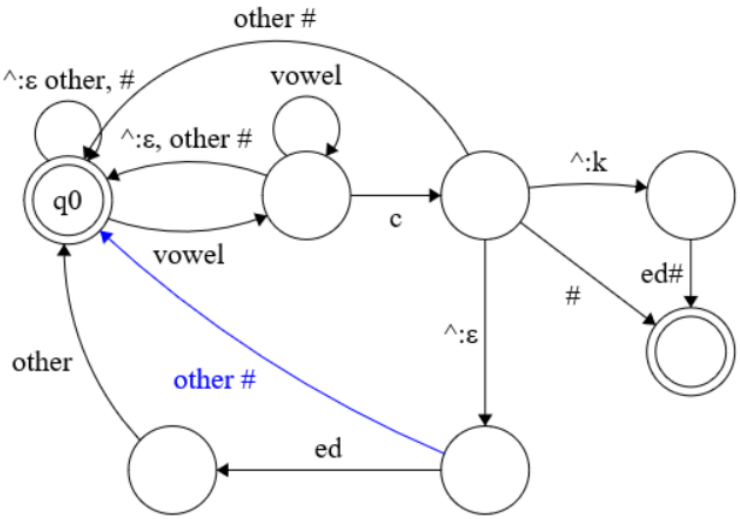
c.

Q3. [5 points]Write the regular expressions that correspond to the following FSAs

Left, Answer: “a(baa)*ba?”

Right: "(^$)|(1*(01*01*)*)"

Q4. [Required for graduate students only - 5 points] Write a Finite State Transducer for the K insertion spelling rule in English.



A:

Q5. [10 points] The Soundex algorithm (Odell and Russell, 1922; Knuth, 1973) is a method commonly used in libraries and older Census records for representing people's names. It has the advantage that versions of the names that are slightly misspelled or otherwise modified (common in hand-written census records) will still have the same representation as correctly-spelled names. (e.g., Jurafsky, Jarofsky, Jarovsky, and Jarovski all map to J612). The rules are as follows:

a. Keep the first letter of the name, and drop all occurrences of non-initial a, e, h, i, o, u, w, y

b. Replace the remaining letters with the following numbers:

b, f, p, v → 1

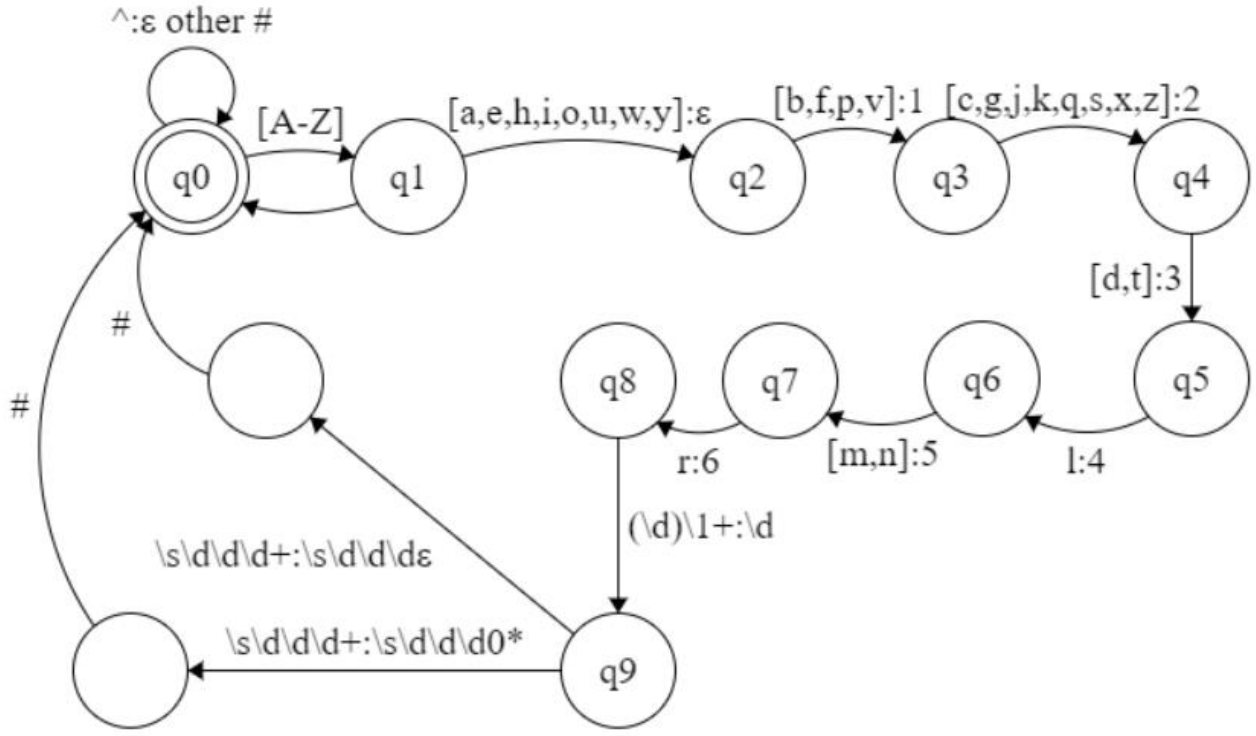
c, g, j, k, q, s, x, z → 2

d, t → 3, l→4, m, n → 5, r→6

c. Replace any sequences of identical numbers, only if they derive from two or more letters that were *adjacent* in the original name, with a single number (i.e., 666 → 6).

d. Convert to the form Letter Digit Digit Digit by dropping digits past the third (if necessary) or padding with trailing zeros (if necessary).

Write a FST (with intermediate tapes if necessary) to implement the Soundex algorithm.



Q6. [5 points] Compute the minimum edit distance between `drive` and `brief`. Show all your work, including the table and the backtrace arrows.

First, define minimum edit distance between drive and brief as D(5,5).

Using Dynamic Programming definition:

$$D[i,j] = \min \begin{cases} D[i-1,j] + \text{del-cost}(source[i]) \\ D[i,j-1] + \text{ins-cost}(target[j]) \\ D[i-1,j-1] + \text{sub-cost}(source[i], target[j]) \end{cases}$$

Levenshtein Distance Convention

$$D[i,j] = \min \begin{cases} D[i-1,j] + 1 \\ D[i,j-1] + 1 \\ D[i-1,j-1] + \begin{cases} 2; & \text{if } source[i] \neq target[j] \\ 0; & \text{if } source[i] = target[j] \end{cases} \end{cases}$$

Minimum Edit Distance Between drive and brief is 4.

Src\Tar	#	d	r	i	v	e
#	0	← 1	← 2	← 3	← 4	← 5
b	↑ 1	↖↑ 2	↖↑ 3	↖ 4	↖ 5	↖ 6
r	↑ 2	↖↑ 3	↖ 2	← 3	← 4	← 5
i	↑ 3	↖ 4	↑ 3	↖ 2	← 3	← 4
e	↑ 4	↖ 5	↑ 4	↑ 3	↖ 4	↖ 3
f	↑ 5	↖ 6	↑ 5	↑ 4	↖ 5	↑ 4

Part II : Language Models

Language identification is the problem of taking a text in an unknown language and determining what language it is written in. N-gram models provide a very effective solution for this problem. For training, use the English (`EN.txt`), French (`FR.txt`), and German (`GR.txt`) texts made available in blackboard/piazza. For test, use the file `LangID.test.txt`. For each of the following problems, the output of your program has to contain a list of [line_id] [language] pairs:

ID LANG Package Dependency: numpy

1 EN
2 FR
3 GR

#**To run my scripts with one command, please use cmd:**

bash run_all_python_scripts_for_HW1.sh

Q7. [10 points] Implement a letter bigram model, which learns letter bigram probabilities from the training data. A separate bigram model has to be learned for each language (from each of the training files provided). Apply the models to determine the most likely language for each sentence in the test file (that is, determine the probability associated with each sentence in the test file, using each of the four language models).

A:
1.Training Data and Test Data has to be in the same directory.
2. Run: python Letter_Bigram_Model.py No Input Parameter Needed.
3. Output file can be found as “Results_Letter_Bigram_Model.txt”

Q8. [5 points] Implement a word bigram model, which learns word bigram probabilities from the training data. Again, a separate model will be learned for each language. Use Add-One smoothing to avoid zero-counts in the data. Apply the models to determine the language ID for each sentence in the test file.
“Add-One smoothing was applied”

A:
In here, I add one more smoothing or method to calculate probability that a word pair that never seen before.
Simply: Log(Word_Pair_ Never_Seen) = log(1/len(Vocabulary)**2), this idea is similar to good turing smoothing. And is very useful to eliminate the scenario “all equal zero if one word pair never seen in a sentence.”

1.Training Data and Test Data has to be in the same directory.
2. Run: python Word_Bigram_Model.py No Input Parameter Needed.
3. Output file can be found as “Results_Word_Bigram_Model.txt”

Q9. [10 points] Implement a word bigram model, which learns word bigram probabilities from the training data. Again, a separate model will be learned for each language. Use Good Turing smoothing to avoid zero-counts in the data. Apply the models to determine the language ID for each sentence in the test file.

A: Add Good Turing Smoothing: Only use recount to get all unseen count = N1/N, and each time, if any unseen word were detected in a word pair, the log probability of this sentence will add log(p_unseen), p_unseen= N1/N**2.
However, this smoothing method doesn't provide a very good results. For word bigram model, if the training set was not large enough. The 2D bigram raw count we built will be inevitable sparse and contain many zero values. That also means, the p_unseen here will also be unreasonable large than other cases. And for a long sentence with many unseen word pairs, its total probability could be given a biased value.

1.Training Data and Test Data has to be in the same directory.
2. Run: python Word_Bigram_Model_Good_Turing_Smoothing.py No Input Parameter Needed.
3. Output file can be found as "Results_Word_Bigram_Good_Turing_Smoothing_Model.txt"

Q10. [required for Graduate students only 10 points] Implement a word Trigram model, which learns word trigram probabilities from the training data. Again, a separate model will be learned for each language. Use either Kneser-Ney smoothing or Katz Back-off (as described in chapter 4 of Jurafsky & Martin).
A: Trigram Model is kind tricky to implement, such as training in EN.txt, to build a trigram model, for unique 4,402 words in my vocabulary for raw trigram counts. It needs 4.4k**3 / 1024**3 ~= 79GB memory, which actually triggers a technical error in my python.

"`Unable to allocate array with shape (4402, 4402, 4402)`"
To solve this problem, you have to use data structure as dictionary in python and treat three words as a set. In this way, it is avoidable for requiring such huge memory usage.

1.Training Data and Test Data has to be in the same directory.
2. Run: python Word_Trigram_Model.py No Input Parameter Needed.
3. Output file can be found as "Results_Word_Trigram_Model.txt"

[5 points] Report the accuracy (#of correctly identified pairs/Total) for each language model above using the gold labels in `LangID.gold.txt`

`#A summary file of all results can be found at "All_Models_Summary.txt"`

Model:	Letter Bigram	Word Bigram With add-one and xxx	Word Bigram With Good Turing Smoothing	Trigram Model
Accuracy Compare with Reference:	86.7%	97.3%	92%	77.33%

Reference:

1. Speech and Language Processing An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition Third Edition draft, and Second Edition
2. Regular Expression on dates (https://www.regular-expressions.info/dates.html)
3. Finite State Machine Designer (http://madebyevan.com/fsm/)
4. Python, Documentation 3.7
5. Numpy
6. Slides from Course