Instructions

1. This note book contains instructions for **COMP9318-Project**.
2. You are required to complete your implementation in a file `submission.py` provided along with this notebook.
3. You are not allowed to print out unnecessary stuff. We will not consider any output printed out on the screen. All results should be returned in appropriate data structures via corresponding functions.
4. You can submit your implementation for **Project** via following link: http://kg.cse.unsw.edu.au/submit/ (http://kg.cse.unsw.edu.au/submit/) .
5. For each question, we have provided you with detailed instructions along with question headings. In case of any problem, you can post your query @ Piazza.
6. You are allowed to add other functions and/or import modules (you may have for this project), but you are not allowed to define global variables. **Only functions are allowed** in `submission.py` .
7. You should not import unnecessary modules/libraries, failing to import such modules at test time will lead to errors.
8. We will provide immediate feedback on your submission. You can access your scores using the online submission portal on the same day.
9. For **Final Evaluation** we will be using a different dataset, so your final scores may vary.
10. You are allowed to have a limited number of Feedback Attempts **(15 Attempts for each Team)**, we will use your **LAST** submission for Final Evaluation.

Q1: Viterbi algorithm for Address Parsing (50 Points)

In this question, you will implement the Viterbi Algorithm for parsing Australian address datasets. Specifically, you are required to write a method `viterbi_algorithm()` in the file `submission.py` that takes THREE arguments as input:

1. **State_File**
2. **Symbol_File**
3. **Query_File**

The files (**State_File**, **Symbol_File**, and **Query_File**) are explained in the following sub-section.

State File:

The **State_File** is a plain text file, where each line carries the information specified below:

- The first line is an integer N, which is the number of states.

- The next N lines are the descriptive names of the states with $ID\ i (0 \le i \le N-1)$. Note that the name of a state is mainly for better readability, as the algorithm distinguishes states by their IDs (except the two special states).
- The next lines are the frequency of **transitions** between two states. Each line contains three fields (denoted as f_1, f_2, f_3), separated by white spaces, meaning that we have seen f_3 number of times where state f_1 transit to state f_2. Note that f_3 could be any non-negative integer ($i.e.$, including 0). Also note that if some state transitions are not specified in this file, we assume the frequency is 0.

Symbol File:

The **Symbol_File** is a plain text file, where each line carries the information specified below:

- The first line is an integer M, which is the number of symbols.
- The next M lines are the descriptive names of the symbols with $ID\ i (0 \le i \le M-1)$. Note that the name of a symbol is important and needed when parsing the addresses, however, only the ID is used in the rest of the program. All symbol names are case-sensitive.
- The next lines are the frequency of **emissions** between a state and a symbol. Each line contains three fields (denoted as f_1, f_2, f_3), separated by white spaces, meaning that we have seen f_3 number of times where state f_1 emits the symbol f_2. Note that f_3 could be any non-negative integer ($i.e.$, including 0). Also note that if some state-symbol emissions are not specified in this file, we assume the frequency is 0.

Query File:

The **Query_File** consists of one or more lines, where each line is an address to be parsed.

1. Parsing. To parse each query line, you first need to decompose the line into a sequence of tokens with the following rules:
2. Whenever there is one or more white spaces, you can break there and discard the white spaces.
3. We take the following punctuations as a state, so you need to correctly extract them.
   *, ( ) / - & *

For example, the following line:

8/23-35 Barker St., Kingsford, NSW 2032

will be parsed into the following tokens (one token a line)

8
/
23
-
35
Barker
St.
,
Kingsford
,
NSW
2032

Then you can convert each token into the symbol IDs, with the possibility that some symbols are never seen in the `Symbol_File`, and you need to treat it as a special `UNK` symbol.

Smoothing

Smoothing the Transition Probabilities.

We distinguish two kinds of states:

1. Special states: i.e., the `BEGIN` and `END` states. Their state names are exactly `BEGIN` and `END` .
2. Normal states: the rest of the states in the `State_File` .

The maximum likelihood estimate for the transition probability $A[i, j]$ is $A[i, j] = \frac{n(i,j)}{n(i)}$, where $n(i, j)$ is the frequency of seeing state sequence i followed by j, and $n(i)$ is the frequency of seeing the state i. This estimate will assign probability 0 to any state transition not seen in the training data. To avoid this problem, we use the simple `add-1 smoothing` .

In our project, there will be N states (including the special states). We use the following rules:

- For the `BEGIN` state, there is no transition to it, i.e., the probability is indeed 0.0.
- For the `END` states, there is no transition from it, i.e., the probability is indeed 0.0.
- Otherwise, we compute $A[i, j]$ as $A[i, j] = \frac{n(i,j)+1}{n(i)+N-1}$
  tips: $N - 1$, because there are at most $N - 1$ states that can be the j.

Smoothing the Emission Probabilities.

We distinguish two kinds of symbols:

1. Known symbols (which are symbols that appeared in the `Symbol_File`), and
2. Unknown symbols (i.e., symbols that never appeared in the `Symbol_File`), which are treated as if they are the same special symbol `UNK` .

We use the following rules:

- For the `BEGIN` and `END` states, they do not emit any symbol. We specify that every state sequence must start in the `BEGIN` state and end in the `END` state. Therefore, the state transition probabilities from the `BEGIN` state and those to the `END` state need to be considered. For example, the generation probability of an observation sequence o_1 with state sequence $[BEGIN, 1, END]$ is $A[BEGIN, 1] \cdot B[1, o_1] \cdot A[1, END]$. The transition probabilities from the `BEGIN` state can be deemed as equivalent to the initial probability distribution π in the standard HMM literature.
- For the other states:
  - If the symbol o_j is a known symbol, its emission probability from state S_i after smoothing is $B[i, j] = \frac{n(i,j)+1}{n(i)+M+1}$, where $n(i)$ is the total number of times state i has been seen, and M is the number of sybmols in the `Symbol_File` .
  - If the symbol is an unknown symbol, its its emission probability from state S_i after smoothing is $B[i, j] = \frac{1}{n(i)+M+1}$.

tips: $M + 1$, because of the UNK token.

Output Format (Q1)

If there are t queries in the **QUERY_File**, you need to output a list consisting of t sublists, where each sublist gives the most likely state sequence for the corresponding address and its log probability (given the observed address and the HMM model). You should append the log probability for each sequence at the end of the list.

Specifically, the format of each sublist is:

- The state sequence consists of L state IDs.
- The log probability is the natural logarithm of the probability, $i.e.$, $\ln(p)$. You should output all its digitals – do **not** truncate it.

Q2 Extending Viterbi for top-k Parsing (30 points)

The standard Viterbi algorithm ($Q1$) gives the state sequence that has the maximum probability of generating the observed symbol sequence. In $Q2$, you need to extend the algorithm to generate the best k state sequences (in terms of the probability of generating the observed symbol sequence), where k is an integer parameter corresponding to the `top-k` results to be returned by the model.

Specifically, you are required to write a method `top_k_viterbi()` in the file `submission.py` that takes `FOUR` arguments as input:

- **State_File**
- **Symbol_File**
- **Query_File**
- **k** *(top-k state sequence to be output for each address in the* *Query_File**)*.

The format of the files (**State_File**, **Symbol_File**, and **Query_File**) is same as explained in $Q1$.

Output Format (Q2)

If there are t queries in the **QUERY_File**, you need to output a list consisting of $t * k$ sublists, where first k sublsists gives the most likely state sequence for the first address and its log probability, followed by k sublists for second address, and so on. The format of each sublist is same as described previously for $Q1$, i.e.,

- The state sequence consisting of L state IDs.
- The log probability, $i.e.$, $\ln(p)$.

Ties

Whenever two different state sequences have the same probability, we break the tie as follows:

- We look at the last state, and prefer the sequence with a smaller state (in terms of state ID).
- If there is a tie, we move to the previous state and repeat the above comparison until we resolve the tie.

For example, assume the following two state sequences have the same highest probability, the second is preferred, i.e., selected as the output).

1 2 3 4
2 1 3 4

Assumptions: You can always assume the model files are correct, i.e., you do not need to deal with possible errors/inconsistencies in the file.

Evaluation

Your results will be compared with the Ground Truth labels to compute `Total Number of Incorrect Labels` for all the test instances.

For example, we have 3 test cases (**Note:** In this example, we ignore the log probability in Prediction just to facilitate demonstration) as follows:

Prediction = $[[0, 3, 2, 4, 5, 1], [1, 3, 4, 2, 5, 4], [5, 4, 1, 2, 3]]$
Ground_Truth = $[[1, 3, 2, 5, 4, 1], [1, 3, 6, 2, 5, 4], [3, 4, 1, 2, 3]]$

`Total Number of Incorrect Labels` $= 3 + 1 + 1 = 5$

Note:

- For $Q1$ and $Q2$, we will evaluate your submission based on the correctness of your implementation. For these questions, you are only required to correctly implement the viterbi algorithm using `add-1 smoothing` ($Q1$) and later extend it for `top-k` Parsing($Q2$).
- In $Q3$, you are required to design a methodology that can improve the performance of the model (i.e., reduce the number of incorrect labels) compared to that of $Q1$.

Q3 Advanced Decoding (20 points)

For this part, you are required to design a methodology that can reduce the `Total Number of Incorrect Labels` in $Q1$ by a `margin` $>$ `15` on the given `dev set` (i.e., `Query_File` with corresponding Ground truth in the file `Query_Label`).

You should complete your implementation for $Q3$ in the method `advanced_decoding()` in the file `submission.py` . Its input parameters and output format are same as that of `viterbi_algorithm()` in $Q1$.

Note:

- For $Q3$, you should design a robust methdology for reducing the `Total Number of Incorrect Labels` . Your method should not overfit the provided `dev set` . Your approach should be able to perform equally well for
  - `dev set` used in the Feedback system,
  - `test set` to be used for final evaluation.

Output Format(Q3)

The output format for $Q3$ is similar to that of $Q1$

How we test your implementation...

In [4]:

```python
import submission as submission


# How we test your implementation for Q1
# Model Parameters...
State_File ='./toy_example/State_File'
Symbol_File='./toy_example/Symbol_File'
Query_File ='./toy_example/Query_File'
viterbi_result = submission.viterbi_algorithm(State_File, Symbol_File, Query_Fi


# How we test your implementation for Q2
# Model Parameters...
State_File ='./toy_example/State_File'
Symbol_File='./toy_example/Symbol_File'
Query_File ='./toy_example/Query_File'
k = 5 #(It can be any valid integer...)
top_k_result = submission.top_k_viterbi(State_File, Symbol_File, Query_File, k)



# How we test your implementation for Q3.
# Model Parameters...
State_File ='./toy_example/State_File'
Symbol_File='./toy_example/Symbol_File'
Query_File ='./toy_example/Query_File'
advanced_result = submission.advanced_decoding(State_File, Symbol_File, Query_F

# Example output for Q1.
for row in viterbi_result:
    print(row)
```

```
[3, 0, 0, 1, 2, 4, -9.843403]
[3, 2, 1, 2, 4, -9.397116]
```

Project Submission and Feedback

For project submission, you are required to submit the following files:

1. Your implementation in a python file `submission.py` .
2. A report `Project.pdf` You need to write a concise and simple report illustrating

   - Implementation details of $Q1$.
   - Details on how you extended the Viterbi algorithm ($Q1$) to return the top-k state sequences (for $Q2$).
   - Your approach for the advanced decoding ($Q3$).

Note: Every team will be entitled to **15 Feedback Attempts** (use them wisely), we will use the last submission for final evaluation.

4. Bonus Part (20 points)

After completing the project, you are welcomed to design/implement a methodology that

1. Significantly outperforms your implementation ($Q1$ and $Q3$), i.e., reduces the `Total Number of Incorrect Labels` on the given `dev set` (`Query_File` with corresponding Ground truth shared in

the file `Query_Label`) by a `margin` ≥ 17.
2. Should be executable.

Submission of Bonus Part

If you opt for the bonus part, you need to seperately submit a `.zip` file $(< 50MB)$ which contains:

1. The code illustrating your implementation
2. A `.pdf` report

The report should at least contain the following parts:

1. The implementation detail of your model (e.g., what are the major differences relative to $Q1$ and $Q3$.
2. The instruction of how to execute your code.

NOTE:

- We will award BONUS marks to `Top-20` best-performing teams.
- We will `NOT` provide any `FEEDBACK` for the `BONUS Part`.
- If your code fails to execute, NO BONUS point will be awarded.
- You should design a robust methdology for reducing the `Total Number of Incorrect Labels`. Your method should not overfit the provided `dev set`. You should be able to perform equally well on `test set` to be used for final evaluation.

```
In [ ]:
```