HW 6

Presented by: Aditya Patel and Athar Sefid

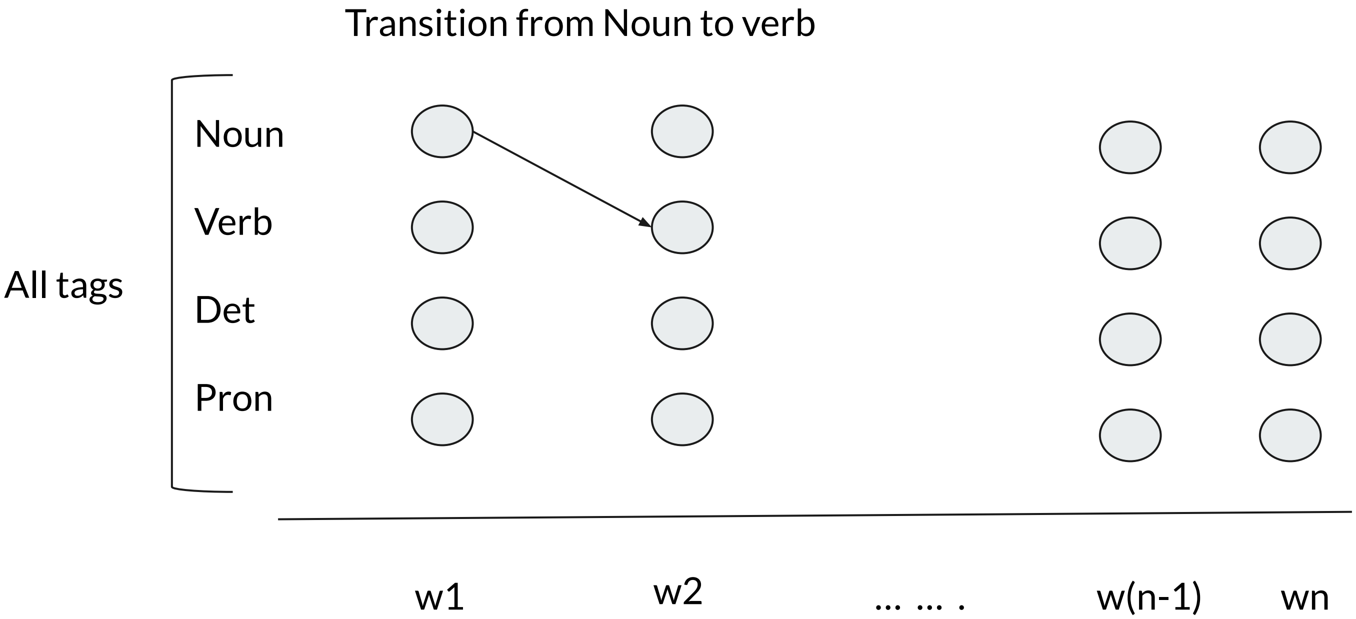
HW 6 Question 2

- Need to implement __init__(self, sentence) for Tagger class.
- The following internal variables need to be initialized:
  - self.tag: **set** of tags (VERB, NOUN etc.)
  - self.init_probs: self.init_probs[t] = (log) probability (with laplace smoothing) that a sentence begins with tag t
  - self.trans_probs: self.trans_probs[(t_i, t_j)] = (log) probability (with laplace smoothing) that a tag t_i occurs before t_j
  - self.em_probs: self.em_probs[(t_i, w_j)] = (log) probability (with laplace smoothing) that a token w_j is generated given tag t_i
- Data Structure: Nested Dictionary
  - A[x][y]

HW 6 Question 3

- Find the maximum emission probability among all tags for each token:
  - for token in tokens:
    - // find tag **tag** for given token token such that emission_probability(**tag**, token) is maximized
    - for tag in tags:
      - Self.em_probs[**tag**][token]
    - // easy to do with lambda function passed to in-built max() function
- Can do it in one line: return [<list-comprehension>].

Trellis for POS tagging



Viterbi recursion

- **Viterbi recursion computes the *maximum probability* path to state *j* at time *t* given that the partial observation $o_1 \ldots o_t$ *has been* generated**

$$v_t(\mathrm{j}) = max_{i=1}^{N}\ v_{t-1}(\mathrm{i}) a_{ij} b_j(o_t)$$

$v_{t-1}(i)$	the **previous Viterbi path probability** from the previous time step
a_{ij}	the **transition probability** from previous state q_i to current state q_j
$b_j(o_t)$	the **state observation likelihood** of the observation symbol o_t given the current state j

Viterbi Algorithm

- **Initialization:** $\delta_1(i) = \pi_i b_j(o_1) \quad 1 \le i \le N$
- **Induction:**

$$\delta_t(j) = \left[\max_{1\le i\le N} \delta_{t-1}(i)\, a_{ij}\right] b_j(o_t) \qquad 2 \le t \le T, 1 \le j \le N$$

$$\psi_t(j) = \left[\arg\max_{1\le i\le N} \delta_{t-1}(i)\, a_{ij}\right]$$ ***(Backpointers)***

- **Termination:** $q_T^* = \arg\max_{1\le i\le N} \delta_T(i)$ ***(Final state!)***

- **Backpointer path:** $q_t^* = \psi_{t+1}(q_{t+1}^*) \quad t = T-1,\ldots,1$

Pseudo code

```
initialization:
For t in tags:
      Delta[0][tag] = prob(t being starting tag) * prob(w1 comes with tag t)
For j in range( 1, len(text)):
     For curtag in tags:
      bestprob, bestPrevTag
      For prevtag in tags:
           Prob = delta[j-1][prevtag] * transit(prevtag, curtag)
           // update the best
      //updata delta[tag][t]
      //keep track of previous best tags
```

HW7

Sudoku problem

1-Fill each square with a digit from 1 to 9.
2-Each row, column, and block must contain each digit exactly once.

	1	5		2				9
	4					7		
	2	7			8			
9	5				3	2		
7								6
		6	2				1	5
			6			9	2	
		4					8	
2				3		6	5	

Basic Functions

- read_board(path)
    - Dictionary :
        - key:(row,col)
        - Value: set of all possible values
-
- get_values(self, cell)
    - get_values((0,0)) -> set([1, 2, 3, 4, 5, 6, 7, 8, 9])
    - get_values((0,1)) -> set([1])
- sudoku_cells():
    - Returns the list of all cells in a Sudoku puzzle as (row, column) pairs.
- sudoku_arcs():
    - returns the list of all arcs between cells in a Sudoku puzzle corresponding to inequality constraints.

```
* 1 5 * 2 * * * 9
* 4 * * * * 7 * *
* 2 7 * * 8 * * *
9 5 * * * 3 2 * *
7 * * * * * * * 6
* * 6 2 * * * 1 5
* * * 6 * * 9 2 *
* * 4 * * * * 8 *
2 * * * 3 * 6 5 *
```

Textual Representation

Easy difficulty solutions

- remove_inconsistent_values(self, cell1, cell2) :
  - removes any value in the set of possibilities for cell1 for which there are no values in the set of possibilities for cell2 satisfying the corresponding inequality constraint.
  - cell1->(1,2,3) and cell2->(1,3,5) : NO inconsistency
  - cell1->(1,2,3) and cell2->(1) : cell1->(2,3) and cell2->(1)

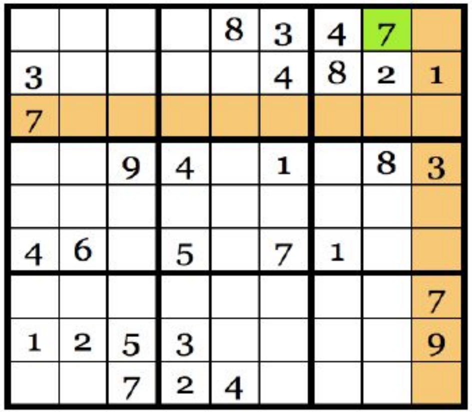
- infer_ac3(self)

```
While Sudoku.ARCS:
        Cell1, cell2 = Sudoku.ARCS.pop()
        remove_inconsistent_values(cell1, cell2)
        If cell1 possible values is changes:
                Update Sudoku.ARCS
```

Medium difficulty

infer_improved:

- while new assignment:
- self.infer_ac3()
- for cell in Sudoku.CELLS:
- for value in board[cell]:
- if unique_in_row or
- Unique_in_col or
- Unique_in_block:
- self.board[cell] = set([value])



Inference Beyond AC-3

Hard problems

- infer_with_guessing(board):
-     infer_improved()
-     for cell in CELLS:
-         for value in board[cell]:
-             infer_with_guessing(newboard)