# 50.007 Machine Learning

# Design Project

Samuel Lim 1000971

Lin Junsheng 1000947

Shaun Chiang 1001085

## Part 2

## Emission Parameters

In part 2, we are interested in obtaining the emission parameters from the training set using the maximum likelihood estimation.

However, we still need to first parse in the data. We do this in parse\_labeled\_data(), and returns a 2-d integer array (n columns with 7 rows, each row representing a label) along with a list of words corresponding to each column. From this, we are able to obtain the count of each label for a specific word.

Using this 2-d integer array, we then are able to calculate emission parameters using the above formula, for the test set dev.in. We use function emission(), which outputs a Collections.OrderedDict() with the word as key, and list of the 7 label emission parameters. We note that some words that appear in the test set may not appear in the training set; we simply assign a fixed probability to all new words, using the formula described:

* If X is a new word:
* If x appeared in the training set:

From this, we get dev.p2.out by choosing the label with the highest emission parameter, and assigning it to the word. From here, we calculate the various scores required:

1. Results

**SG**

Entity in gold data: 4779

Entity in prediction: 5978

Correct Entity: 641

Entity precision: 0.1072

Entity recall: 0.1341

Entity F: 0.1192

Correct Sentiment: 164

Sentiment precision: 0.0274

Sentiment recall: 0.0343

Sentiment F: 0.0305

**CN**

Entity in gold data: 935

Entity in prediction: 1665

Correct Entity: 64

Entity precision: 0.0384

Entity recall: 0.0684

Entity F: 0.0492

Correct Sentiment: 20

Sentiment precision: 0.0120

Sentiment recall: 0.0214

Sentiment F: 0.0154

**ES**

Entity in gold data: 1326

Entity in prediction: 2493

Correct Entity: 138

Entity precision: 0.0554

Entity recall: 0.1041

Entity F: 0.0723

Correct Sentiment: 31

Sentiment precision: 0.0124

Sentiment recall: 0.0234

Sentiment F: 0.0162

**EN**

Entity in gold data: 662

Entity in prediction: 1405

Correct Entity: 98

Entity precision: 0.0698

Entity recall: 0.1480

Entity F: 0.0948

Correct Sentiment: 30

Sentiment precision: 0.0214

Sentiment recall: 0.0453

Sentiment F: 0.0290

## Part 3

For part 3, we were tasked to calculate the transmission parameters as well as run the Viterbi algorithm with the emission and transmission parameters

1. Transmission Parameters

For transmission parameters, we first spilt our training data set into its respective sequences.

After splitting the data set, we create a 7 x 7 2D-array to store the results in the following format.

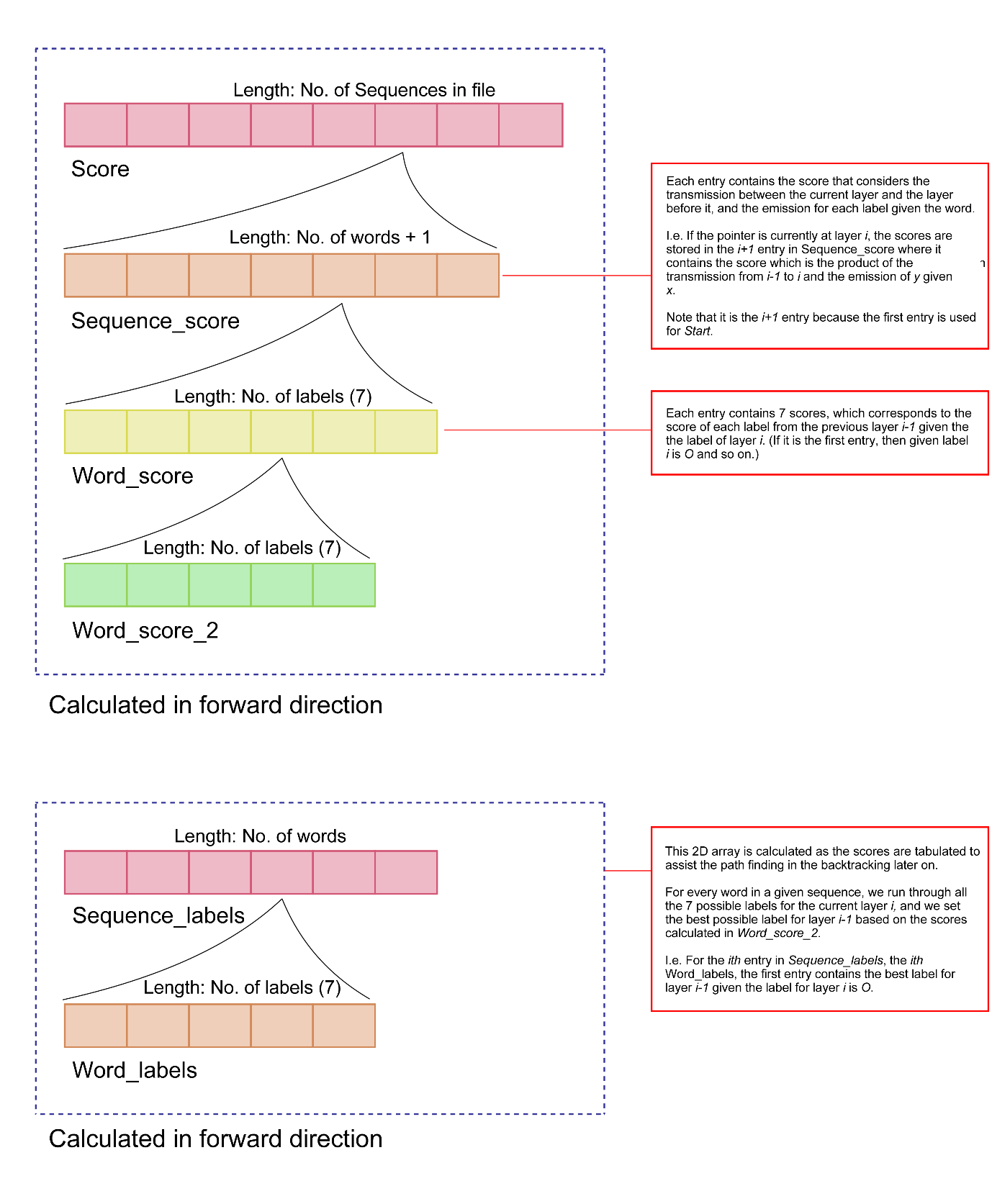
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Next From | O | I-positive | B-positive | I-neutral | B-neutral | I-negative | B-negative | End |
| O |  |  |  |  |  |  |  |  |
| I-positive |  |  |  |  |  |  |  |  |
| B-positive |  |  |  |  |  |  |  |  |
| I-neutral |  |  |  |  |  |  |  |  |
| B-neutral |  |  |  |  |  |  |  |  |
| I-negative |  |  |  |  |  |  |  |  |
| B-negative |  |  |  |  |  |  |  |  |
| Start |  |  |  |  |  |  |  |  |

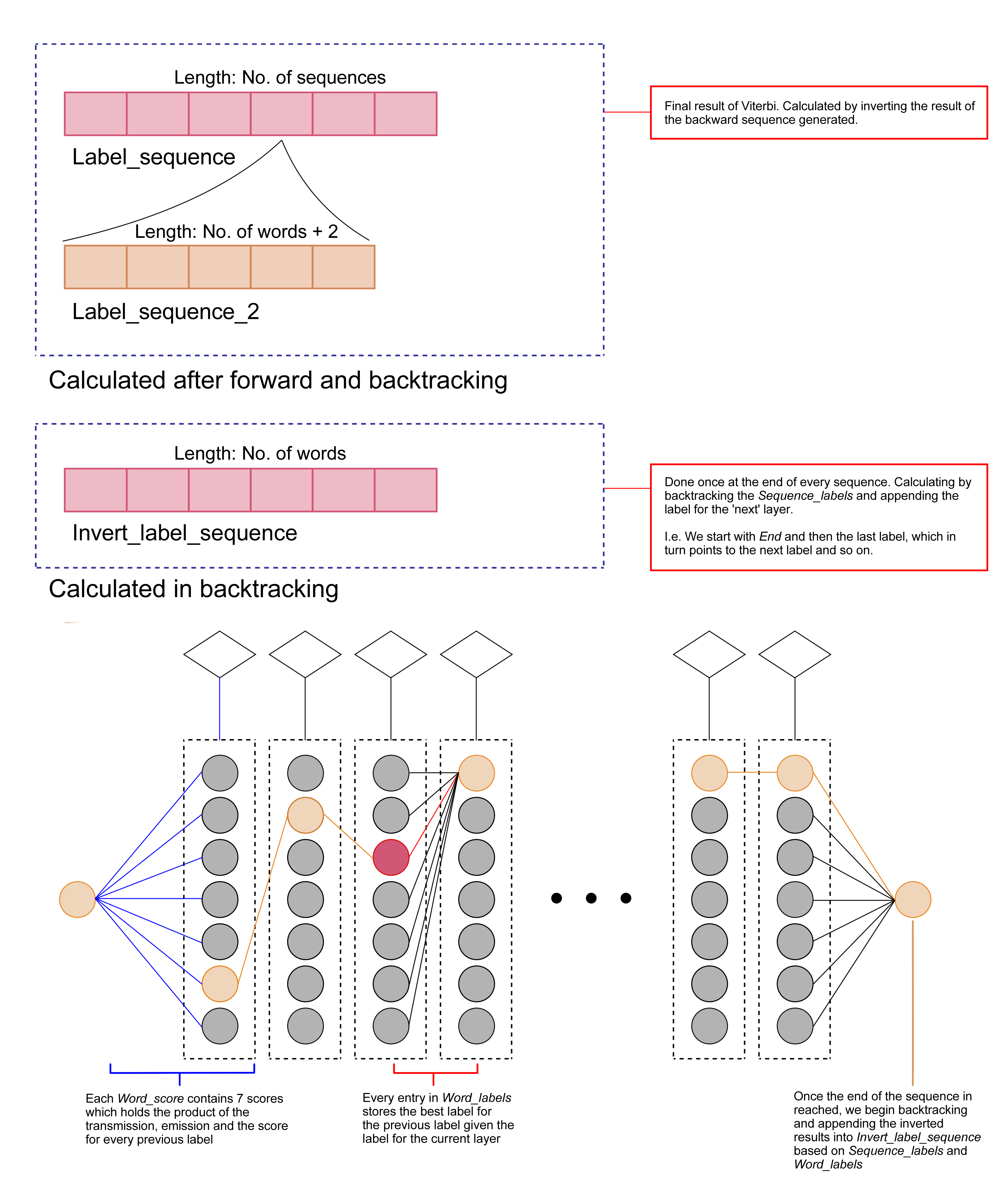
We then start by filling out the count for the table. Each time a case is observed, we increment the count. I.e. if we see a case of O 🡪 B-negative, we increment the table entry of [0, 6] by one. Note that the first dimension of the array is stored as rows, meaning entry [0] is row 1, entry [3] is row 4 etc.

Once the table is complied, we calculate the sum of each row, and then calculate a new 7 x 7 2D array, this time containing the transmission parameters using:

1. Viterbi

For Viterbi, we used the following structure.





We generated a 4D array where the structure of the array is as follows:

* 1st Dimension: List of all sequences
* 2nd Dimension: List of all words in each sequence
* 3rd Dimension: List of all labels in each word
* 4th Dimension: List of each possible combination with the labels of the previous layer per label

**Start**

At the start of every sequence, we only calculate up to the 3rd dimension, and add the scores based on and . We multiply the two scores and add it to the array.

**Middle layers**

We then run through each label, with seven in total, with the seven labels of the previous layer. We multiply the score from each label from the previous layer with the transmission and emission score of the current layer, and store it in the 4th dimension of the array. This will give us a total of 49 scores per layer (sum of 3rd and 4th dimension). After running through all the seven labels of the previous layer, we compare the best label for layer given label for layer .

Once we finish running through all 49 possible combinations, and have the 7 best label for layer given each label in layer as well as their respective scores then append the label of into the selected sequence.

**Note:** that the pointer is at layer but we are appending the label for layer .

**End**

At the end, we do something similar to the middle layers, but we only calculate till the 3rd dimension. We then add the final label of the sequence and then add an ‘End’ label as well.

1. Results

**SG**

Entity in gold data: 4779

Entity in prediction: 4041

Correct Entity: 301

Entity precision: 0.0745

Entity recall: 0.0630

Entity F: 0.0683

Correct Sentiment: 132

Sentiment precision: 0.0327

Sentiment recall: 0.0276

Sentiment F: 0.0299

**CN**

Entity in gold data: 935

Entity in prediction: 1660

Correct Entity: 20

Entity precision: 0.0120

Entity recall: 0.0214

Entity F: 0.0154

Correct Sentiment: 6

Sentiment precision: 0.0036

Sentiment recall: 0.0064

Sentiment F: 0.0046

**ES**

Entity in gold data: 1326

Entity in prediction: 2262

Correct Entity: 98

Entity precision: 0.0433

Entity recall: 0.0739

Entity F: 0.0546

Correct Sentiment: 42

Sentiment precision: 0.0186

Sentiment recall: 0.0317

Sentiment F: 0.0234

**EN**

Entity in gold data: 662

Entity in prediction: 769

Correct Entity: 33

Entity precision: 0.0429

Entity recall: 0.0498

Entity F: 0.0461

Correct Sentiment: 12

Sentiment precision: 0.0156

Sentiment recall: 0.0181

Sentiment F: 0.0168

## Part 4

For part 4, we were tasked to find the top-k best output sequences by implementing an algorithm making use of the estimated emission and transmission parameters from earlier parts

**Setup:**

Generate the *transmission\_array* and *emission\_array* using the input file functions designed in part 2 and part 3.

Process input file by splitting the sequences up and storing them in 2d *input\_array*

In addition to the input file, *transmission\_array* and *emission\_array*, the function also takes in a K parameter which sets the desired top result.

We will initialize a 2d array *Sets\_of\_Sequence* which will store the output top K’th state sequences for each of the sequence.

**Top-K Viterbi algorithm**

We will be diving into 2 loops.

The first loop iterates through the sequences and initializes *SequenceScores* and *Sequence*.

*SequenceScores* stores the scores for all nodes of the particular sequence we are iterating. Sequence will eventually store the top K’th state sequences for this particular sequence we are iterating. It will be appended to *Sets\_of\_Sequence* at the end of each loop.

The second loop iterates through the words of the sequence. We will initialize the *word\_score* array which will store scores of all nodes concerning the particular word we are iterating.

These 2 loops will allow us to iterate through the words of each sequence.

**Some theory on** **Top-K Viterbi**

What is different in Top-K Viterbi as compared to usual Top-1 Viterbi is that there will not be just *m* (m = 7 in our case, O, I-positive, B-positive, I-neutral, B-neutral, I-negative, B-negative) nodes in each layer or *word\_score* but rather there are *m* x K nodes storing scores for each layer.

When we are going forward with Top-K Viterbi, for each state in the layer such as “I-positive”, we will store K scores instead of one. These K scores are actually the Top K scores selected from the previous layer after accounting for transmission parameters from Stateprev to Statecurrent and emission parameters of emitting word from Statecurrent where Statecurrent is the state of the word of our current iteration. Note that the previous layer consist of K x *m* scores/nodes as we are keeping track of Top K scores for each state in a layer. Therefore we are choosing the K scores out of K x *m* computed scores to be stored in Statecurrent node. The scores are maintained sorted within the node with the parent node for each score.

**Start**

From the first layer to the next layer, there is actually only one possible path to each of the state of the next layer. Therefore we will only have score kept in each of the state of the next layer. However for simplicity, I have decided to maintain the data structure of the *word\_score* by storing K scores in each of the state except they are all the same and have parent node as start.

The computation of the score for 1st layer is just astart,v bv(emitted word) since there is no previous score.

**Middle Layers (General Case)**

Looking at the data structure of *word\_score*:

*word\_score* STORES *word\_score\_state* STORES *word\_score\_state\_K*

*word\_score\_state* stores the set of Top-K scores to this particular state

*word\_score\_state\_K* stores one of the Top-K scores to this particular state. *word\_score\_state\_K* is actually an array that stores 3 information:

[Kth Score, Previous State Index, Previous K Index]

For example, a *word\_score\_state* has the 2nd *word\_score\_state\_K* with [0.2, 2, 0]. This means that for this particular state of this layer has the Top-2 score of 0.2 from the Top-1 score of the 3rd state node of the previous layer.

As mentioned above, for each of the state of the current layer of iteration, we are finding the Top-K scores to the state. We keep track of the Top-K computed scores along with the parent node’s state index and kth index.

The computation of the each selected score is Maxu PrevScore au,v bv(emitted word) where *u* could be any of the K x *m* node in the previous layer storing the scores.

**End**

The last layer is similar to the middle layers except that there is only one state node which is End node which will also store the Top-K computed scores.

Since there is no word emitted from End state, the computation of the each selected score is Maxu PrevScore au,v where *u* could be any of the K x *m* node in the previous layer storing the scores.

**Decoding via backtracking**

For decoding the Top-K sequences, we will perform backtracking from right to left. Starting from End state, choose the Kth highest score in the previous layer. Subsequently, we will always choose the most optimal score as we backtrack because the Kth highest score will yield the kth best score as long as it traversed the most optimal sub-path.

**Results for 5th best outputs**

**ES**

#Entity in gold data: 1326

#Entity in prediction: 2675

#Correct Entity : 496

Entity precision: 0.1854

Entity recall: 0.3741

Entity F: 0.2479

#Correct Sentiment : 240

Sentiment precision: 0.0897

Sentiment recall: 0.1810

Sentiment F: 0.1200

**EN**

#Entity in gold data: 662

#Entity in prediction: 1530

#Correct Entity : 219

Entity precision: 0.1431

Entity recall: 0.3308

Entity F: 0.1998

#Correct Sentiment : 116

Sentiment precision: 0.0758

Sentiment recall: 0.1752

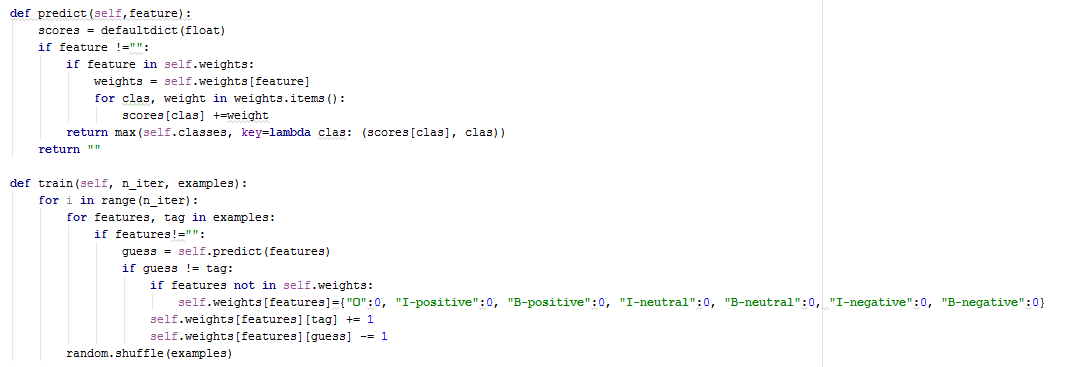
Sentiment F: 0.1058

## Part 5

1. Idea and Implementation

For part 5, we decided to implement a simple ‘perceptron’ version of the HMM problem. Our approach was inspired upon reading Collins’ *Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms*[1]*.* The idea is to adapt a perceptron for every new word recognised, and after training it on some training features and tags, use this information to predict future observations. A perceptron tagger is able to perform comparatively much faster than a standard HMM, and is much more space conserving if the appropriate data structures are chosen. We will investigate the validity of such an approach in terms of accuracy, using the same rubrics of Precision, Recall, and F-score.

To go into more detail, we create class perceptronTagger(), which keeps track of a dictionary of weights. This dictionary then will store the word as key and the various weights for each tag (stored as another dictionary). The training data is then passed through the training algorithm, shown in function train(). After the tagger is trained, it then can be used in function predict() to predict a new set of features.



In train() , we first attempt to guess a tag for the feature based on the current state of our weights. If the guess is correct, we don’t change the state of our weights. However, if the guess is wrong, we add 1 to the weights associated with the correct tag, and we subtract one from the weights associated with the guessed tag. This differs from the conventional idea of the ‘perceptron’ as it does not attempt to converge (it is doubtful that it would be linearly separable as well), but instead, runs through all the given training data *n* number of times, specified by the user. Each time, we shuffle the order of the examples, in order to get different ‘perspectives’ of the data.

In predict(), we use the weights provided from the training data to choose an appropriate tag for each feature. This is simply done by choosing the tag associated to highest weight for a given word.

1. Results

**These given results were done with n=10 iterations.**

**SG**

#Entity in gold data: 4779

#Entity in prediction: 3557

#Correct Entity : 1030

Entity precision: 0.2896

Entity recall: 0.2155

Entity F: 0.2471

#Correct Sentiment : 513

Sentiment precision: 0.1442

Sentiment recall: 0.1073

Sentiment F: 0.1231

**CN**

#Entity in gold data: 935

#Entity in prediction: 1119

#Correct Entity : 258

Entity precision: 0.2306

Entity recall: 0.2759

Entity F: 0.2512

#Correct Sentiment : 167

Sentiment precision: 0.1492

Sentiment recall: 0.1786

Sentiment F: 0.1626

**ES**

#Entity in gold data: 1326

#Entity in prediction: 904

#Correct Entity : 378

Entity precision: 0.4181

Entity recall: 0.2851

Entity F: 0.3390

#Correct Sentiment : 198

Sentiment precision: 0.2190

Sentiment recall: 0.1493

Sentiment F: 0.1776

**EN**

#Entity in gold data: 662

#Entity in prediction: 370

#Correct Entity : 152

Entity precision: 0.4108

Entity recall: 0.2296

Entity F: 0.2946

#Correct Sentiment : 106

Sentiment precision: 0.2865

Sentiment recall: 0.1601

Sentiment F: 0.2054

1. Improvements and Lessons Learnt

We learn that there are possible improvements to be made to our code; one variant that is commonly used and deployed is the averaged perceptron[2], which instead of simply adding or subtracting one from the weights, increments by a weighted average of the weight. The downside of this is that it requires more memory (to keep track of the average) and may perform slower, but with higher generalisability.

Other tweaks we could explore is to pre-process all words in lower-case, to prevent capitalised words registering as a separate word; registering dates or years as a united feature (for example, 1937, 2016, 2002 all become “#YEAR#”), which can be expanded further for phone numbers or addresses, depending on the purpose of the implementation.

Overall, the perceptron approach gives acceptable results, which is great for a relatively simple implementation; with additional tweaks, the perceptron tagger can be used for wide adoption for a variety of projects. For ourselves, we learnt lessons about adapting research paper approaches in our own context; working together in collaborative coding; and learning good coding practices.

**References**

[1] Collins. *Discriminative Training Methods for Hidden Markov Models: Theory and Experiments with Perceptron Algorithms.* EMNLP, July 2002.<http://www.aclweb.org/anthology/W02-1001>

[2] https://explosion.ai/blog/part-of-speech-pos-tagger-in-python