

CS 655: Analyzing Sequences

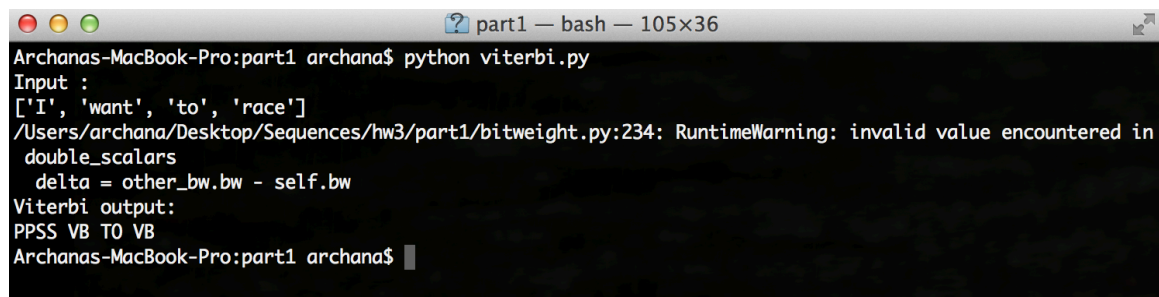
Homework 3

Part 1: Implementing the Viterbi Algorithm

Two matrices of the size $N \times M$ are initialized (N = number of states, M = number of emissions/observations), to store the probabilities and pointers to the state they came from. The product of initial transition probability (transition from $\langle S_0 \rangle$ to that state) and emission probability (for the first word) gives the first column of the probability matrix.

Thereon the value for every cell in the probability matrix is obtained by taking the maximum over all possible paths to the current cell (product of previous column of probability matrix with the corresponding transitional probabilities), and multiplying with the emission probability. At the same time a pointer indicating which cell in the previous column led to the present value is stored in the lookup matrix.

Finally the best path is traced by moving backwards along the lookup table.

A terminal window titled 'part1 — bash — 105x36' on a Mac. The prompt is 'Archanas-MacBook-Pro:part1 archana\$'. The user enters 'python viterbi.py'. The output shows 'Input : ['I', 'want', 'to', 'race']', a 'RuntimeWarning: invalid value encountered in double_scalars' from 'bitweight.py:234', and a line 'delta = other_bw.bw - self.bw'. The final output is 'Viterbi output: PPSS VB TO VB'. The prompt returns to 'Archanas-MacBook-Pro:part1 archana\$' with a cursor.

```
Archanas-MacBook-Pro:part1 archana$ python viterbi.py
Input :
['I', 'want', 'to', 'race']
/Users/archana/Desktop/Sequences/hw3/part1/bitweight.py:234: RuntimeWarning: invalid value encountered in
double_scalars
  delta = other_bw.bw - self.bw
Viterbi output:
PPSS VB TO VB
Archanas-MacBook-Pro:part1 archana$
```

Fig. 1. Sample output for Viterbi algorithm

Part 2: Calculating real-world probabilities

The pre-tagged Wall Street Journal corpus is split into training and testing data (90/10). The training data is modified to get a list of all words and tags separately. Tags of the form $NN|CD$ are split into NN and CD . In order to avoid extra counts for word and tags preceding and following the tags, data is prepared in different ways to calculate emission and transition probabilities.

Input sentence:

[.... dollar/ NN on/ IN foreclosed/ $VB|JJ$ property/ NN ]

Transition probability:

```
[.... dollar/NN on/IN foreclosed/VBN property/NN ....]  
[on/IN foreclosed/JJ property/NN]
```

```
[.... NN IN VBN NN ....]  
[IN JJ NN]
```

Emission probability:

```
[.... dollar/NN on/IN foreclosed/VBN foreclosed/JJ property/NN....]
```

```
[..... ['dollar', 'NN'], ['on', 'IN'], ['foreclosed', 'VBN'], ['foreclosed',  
'JJ'], ['property', 'NN'], ....]
```

While calculating transition probabilities a new line is added to the training data containing a tag preceding and a tag following along with the second part of tag to be split. The original line is retained with the first part of the tag. For calculating emission probabilities, the word is repeated twice in the same sentence with different tags. This is done to avoid excess count of other words ('on' and 'property' in the above example) while calculating emission probabilities. (I guess this is not necessary here because the number for tags of the form NN|CD are very few in this corpus, but might have an effect if the corpus had many words of this form.) In this corpus, training with or without splitting the tag did not have much impact on the accuracy, but it definitely had an impact on the time taken for decoding, which I have explained later.

The transition and emission probabilities using maximum-likelihood estimation (MLE) are calculated using the code provided. While testing the model, all words that are not seen in training corpus are matched to a key 'UNK', and assigned the tag 'NN' as it is the most frequent one. While calculating the transition probabilities, if there was no transition from state x to y, but there is a probability mass on state x, then the transition probability from state x to y is set to be equal to transition for state x to state 'NN'. This is done to avoid all zeros while looking for the best path. Without these two conditions the accuracy while testing 10% of the corpus was as low as 55.58%, but inclusion of these conditions increased the accuracy to 95.69%.

Using Witten-Bell smoothing avoided the need to add the extra condition in calculating transition probabilities, as it took care of assigning probabilities to unseen states. But still the tagging of words out-of-vocabulary had to be done, as emission probabilities cannot be smoothed using Witten-Bell smoothing. For this the data is prepared similar to MLE. The accuracy for 10% testing data without substituting for OOVs is 56.84%, while with OOVs taken care of it increased to 95.91%. Low accuracy is observed with OOV words because, when an OOV word is encountered, all the states before the OOV point to the state zero (which is PRP\$ in this case) while calculating the best path (i.e. all words are tagged as PRP\$)

Sample terminal output showing MLE and WB implementations tagging sentences are shown in fig.2 and 3 respectively.

```

Archanas-MacBook-Pro:output archana$ python mle_final.py
Input sentence:
['In/IN major/JJ market/NN activity/NN :/:n']
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in double_scalars
  delta = other_bw.bw - self.bw
Tags using MLE:
IN JJ NN NN :
Archanas-MacBook-Pro:output archana$ python mle_final.py
Input sentence:
["But/CC several/JJ other/JJ traders/NNS contend/VBP investors/NNS have/VBP overreacted/VBN to/TO junk-bond/NN j
itters/NNS ,/, and/CC that/IN stock/NN prices/NNS will/MD continue/VB to/TO recover/VB ./. ```` They/PRP shot/V
BD the/DT whole/JJ orchestra/NN just/RB because/IN the/DT piano/NN player/NN hit/VBD a/DT bad/JJ note/NN ,/, ''/
'' said/VBD <NNP>/NNP <NNP>/NNP ,/, president/NN of/IN <NNP>/NNP <NNPS>/NNPS <NNP>/NNP ,/, referring/VBG to/TO t
he/DT stock/NN market/NN 's/POS plunge/NN <NNP>/NNP on/IN news/NN of/IN trouble/NN in/IN financing/VBG the/DT <N
NP>/NNP <NNP>/NNP ./. buy-out/NN ./.n"]
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in double_scalars
  delta = other_bw.bw - self.bw
Tags using MLE:
CC JJ JJ NNS VBP NNS VBP VBN TO NN NNS , CC DT NN NNS MD VB TO VB . `` PRP VBD DT JJ NN RB IN DT NN NN VBD DT JJ
NN , '' VBD NNP NNP , NN IN NNP NNPS NNP , VBG TO DT NN NN POS NN NNP IN NN IN NN IN VBG DT NNP NNP . NN .
Archanas-MacBook-Pro:output archana$ python mle_final.py
Input sentence:
["Although/IN stocks/NNS have/VBP led/VBN bonds/NNS this/DT week/NN ,/, some/DT traders/NNS predict/VBP that/IN
relationship/NN will/MD reverse/VB during/IN the/DT next/JJ few/JJ weeks/NNS ./. <NNP>/NNP 's/POS <NNP>/NNP <NNP
>/NNP fears/VBZ a/DT huge/JJ wave/NN of/IN <NNP>/NNP borrowing/VBG early/JJ next/JJ month/NN will/MD drive/VB do
wn/IN <NNP>/NNP bond/NN prices/NNS ./. That/IN ,/, coupled/VBN with/IN poor/JJ third-quarter/NN corporate-earnin
gs/NNS comparisons/NNS ,/, ```` will/MD make/VB trouble/NN for/IN the/DT equity/NN market/NN for/IN the/DT next
/JJ <CD>/CD to/TO <CD>/CD months/NNS ,/, ''/' he/PRP says/VBZ ./.n"]
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in double_scalars
  delta = other_bw.bw - self.bw
Tags using MLE:
IN NNS VBP VBN NNS DT NN , DT NNS VBP DT NN MD VB IN DT JJ JJ NNS . NNP POS NNP NNP VBZ DT JJ NN IN NNP NN RB JJ
NN MD VB IN NNP NN NNS . DT , VBN IN JJ JJ NN NNS , `` MD VB NN IN DT NN NN IN DT JJ CD TO CD NNS , '' PRP VBZ
.
```

Fig. 2. Sample output showing MLE implementation tagging sentences

```

Archanas-MacBook-Pro:output archana$ python wb_final.py
Input sentence:
['In/IN major/JJ market/NN activity/NN :/\n']
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in
double_scalars
    delta = other_bw.bw - self.bw
Tags using WB:
IN JJ NN NN :
Archanas-MacBook-Pro:output archana$ python wb_final.py
Input sentence:
['Houston-based/JJ <NNP>/NNP <NNP>/NNP <NNPS>/NNPS ,/, which/VBD had/VBD sales/NNS of/IN about/IN $/$ <CD>/CD <C
D>/CD last/JJ year/NN ,/, sells/VBZ coffee/NN under/IN the/DT <NNP>/NNP <NNP>/NNP and/CC <NNP>/NNP brands/NNS to
/TO restaurants/NNS ,/, hotels/NNS ,/, offices/NNS and/CC airlines/NNS ./. The/DT acquisition/NN ```` gives/VBZ
us/PRP additional/JJ production/NN capacity/NN for/IN the/DT food/NN service/NN coffee/NN business/NN and/CC a/
DT stronger/JJR distribution/NN network/NN ,/, '``' a/DT <NNP>/NNP spokesman/NN said/VBD ./.n']
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in
double_scalars
    delta = other_bw.bw - self.bw
Tags using WB:
JJ NNP NNP NNPS , WDT VBD NNS IN IN $ CD CD JJ NN , VBZ NN IN DT NNP NNP CC NNP NNS TO NNS , NNS , NNS CC NNS .
DT NN `` VBZ PRP JJ NN NN IN DT NN NN NN NN CC DT JJR NN NN , ' DT NNP NN VBD .
Archanas-MacBook-Pro:output archana$ python wb_final.py
Input sentence:
['Gold/NN ,/, a/DT closely/RB watched/VBN barometer/NN of/IN investor/NN anxiety/NN ,/, was/VBD little/RB change
d/VBN ./. The/DT dollar/NN initially/RB fell/VBD against/IN other/JJ major/JJ currencies/NNS on/IN news/NN that/
IN the/DT <NNP>/NNP trade/NN deficit/NN surged/VBD in/IN <NNP>/NNP to/TO $/$ <CD>/CD <CD>/CD ./. But/CC the/DT d
ollar/NN later/RB rebounded/VBD ,/, finishing/VBG slightly/RB higher/JJR against/IN the/DT yen/NN although/IN sl
ightly/RB lower/JJR against/IN the/DT mark/NN ./.n']
/Users/archana/Desktop/Sequences/hw3/part2/output/bitweight.py:234: RuntimeWarning: invalid value encountered in
double_scalars
    delta = other_bw.bw - self.bw
Tags using WB:
NN , DT RB VBN NN IN NN NN , VBD RB VBN . DT NN RB VBD IN JJ JJ NNS IN NN IN DT NNP NN NN VBD IN NNP TO $ CD CD
. CC DT NN RB VBD , VBG RB JJR IN DT NN IN RB JJR IN DT NN .
Archanas-MacBook-Pro:output archana$

```

Fig. 3. Sample output showing WB implementation tagging sentences

Part 3: Evaluation

The corpus has 45 different tags. The frequency of different tags in the corpus is shown in figure 4.

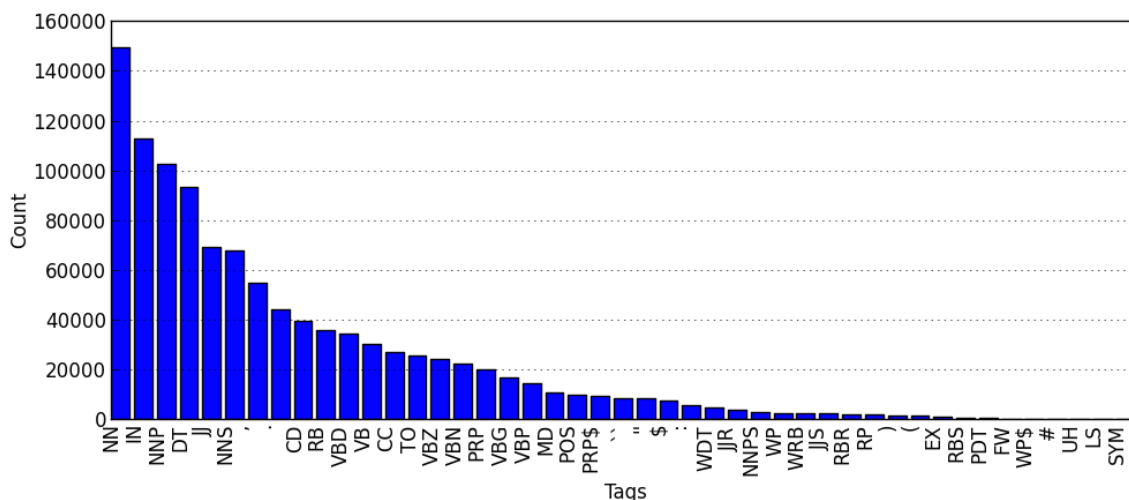


Fig. 4. Frequency of tags in corpus

The testing data is tested for different conditions and accuracy is calculated using the confusion matrix. (Accuracy = Positive instances/All instances)

Table 1. Time taken and accuracy for different testing conditions

		Tag type	Training Time (s)	Testing Time (s)	Accuracy (%)
MLE	Without OOV taken care	NN CD	11.81	294.30	52.22
		NN, CD	12.17	168.14	55.58
	With OOV taken care	NN CD	11.81	317.68	95.69
		NN, CD	12.21	188.92	95.78

Taking care of OOVs improves accuracy to a great extent. Splitting tags results in a slight increase in accuracy, but reduces the time taken by Viterbi algorithm. This is because, if the tags are not split, there are a total of 80 states and for each state, transition from previous 80 states has to be calculated. When tags are split, there are only 45 states; this reduces the time taken by Viterbi algorithm to almost half of what it was previously taking. The increase in training time when tags are split is due to the additional for-loop on all lines in the training data for splitting tags.

Similar effects are seen while processing data using Witten-Bell smoothing, but it results in slightly higher accuracies at 56.84% without taking care of OOV words and 95.91% with OOVs taken care. When the OOV words are matched to 'NN', MLE and WB perform almost similarly (with or without tag separation). Though the results show a slight increase in accuracy, a different split of the corpus results in an accuracy that is slightly higher or lower, but they all fall in the same range (95.5% - 95.9%). Using WB smoothing accuracy is above 95.9% in most of the runs, while MLE touches 95.88% in some runs.

Fig. 5 shows a heat map representation of confusion matrix. Y-axis represents the original tags, while x-axis represents the tags given by Viterbi algorithm. For this representation, tags are separated (NN|CD to NN and CD) in both the training and testing data. Tags are normalized along x-axis (dividing each row by its sum; i.e. values along each original tag sum to 1).

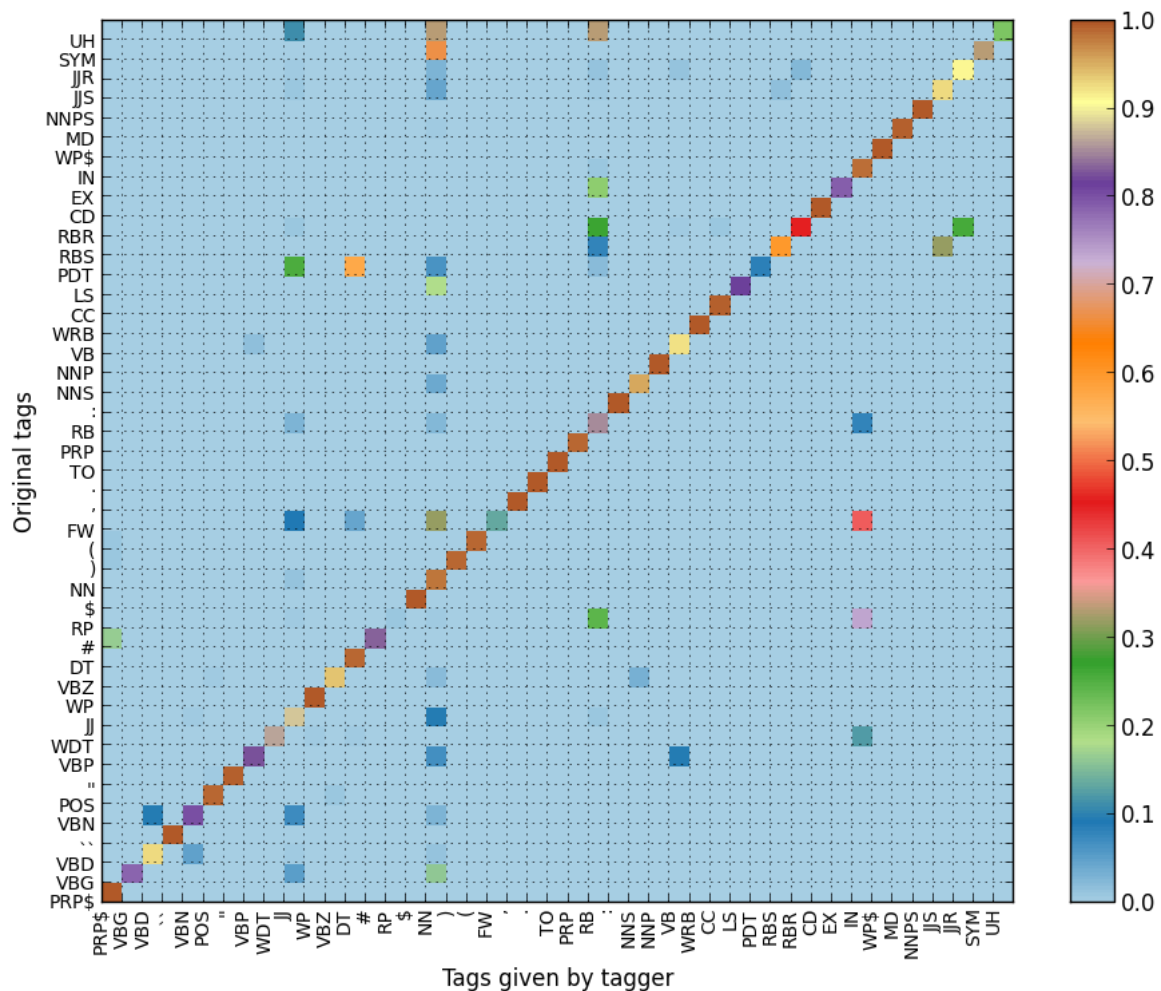


Fig. 5. Heat map representation of Confusion matrix

As all OOV words are matched to NN it can be observed that many words, which are not nouns, are tagged to tag NN. The tags most often identified wrongly are RP (Particle), FW (Foreign word), PDT (Predeterminer) and UH (Interjection).

RP (particle) : RB (Adverb), IN (Preposition)
 FW (Foreign word) : NN (Noun), IN (Preposition), DT (Determiner), JJ (Adjective)
 PDT (Predeterminer) : DT (Determiner), JJ (Adjective), NN (Noun), RB (Adverb)
 UH (Interjection) : RB (Adverb), NN (Noun), JJ (Adjective)

From fig. 5 it can be seen that most of the words the tagger got wrong are tagged to NN (Noun), RB (Adverb), IN (Preposition) or JJ (Adjective).

The tags that are more often confused are VBN-VBD, NN-JJ, VB-VBP, RBS-JJS, RBR-JJR.

VBN-VBD	:	Verb, Past particle – Verb, Past tense
NN-JJ	:	Noun - Adjective
VB-VBP	:	Verb – Verb, single present
RBS-JJS	:	Adverb, superlative – Adjective, superlative
RBR-JJR	:	Adverb, comparative – Adjective, comparative

The most commonly seen mis-tag pattern is words getting tagged to NN. This is mostly due to the condition set in the program to match OOV words to NN. The next most commonly mis-tagged are words that have multiple tags arising from different usage in sentences in the training corpus (eg.: net - NN,JJ; out-IN,RB,RP; next – IN,JJ; only-JJ,RB), are most commonly mis-tagged in the testing corpus. A small part of the wrong-tagged word set is shown below: (word correct_tag tagger_tag)

record JJ NN
 record NN JJ
 record VB JJ
 record VB NN