CS5740: Assignment 3
https://github.com/cornell-cs5740-21sp/a3--group1

Rony Krell rk453 Flora Shen ys2223 Rashmi Sinha rs2584

1 Introduction (5pt)

We are required to implement POS (Part of Speech) tagging. We use WSJ corpus data from the Penn Treebank. Our goal is to achieve a high F1-score in matching generated tags with the held-out tags on the test set

We focus on HMMs (Hidden Markov Models) with the Trellis structure, beam search, and the Viterbi algorithm. We experiment with bigram, trigram and four-gram Viterbi with various smoothing functions and beam search with different beam sizes. Our best results on the development set are 0.9618 for bigrams, 0.9687 on trigrams, and 0.9511, 0.9527, and 0.9525 for beam search with k=4,5,6 respectively. Our best F1 score on the test set is 0.9582 with viterbi trigram.

Dataset	Size	Unique	Total	bigram	trigram	doc length
Train	1,387	37,504	696,475	1344	15375	502.1
Dev	462	20,704	243,021	1164	10943	526
Test	463	20,178	236,582	n/a	n/a	510

Table 1: Size:total number of documents; Unique:number of unique words; Total:number of words. bi/tri-gram:number of bi/tri-gram tags seq; doc length:average length of each documents

2 Data (5pt)

See Table 1, 2 for data statistics.There are 46 tags in the data. The training and development data sets consist of documents words and corresponding tags. All words are preserved with original cases which helps in unknown words handling. We used the whole dataset, and in order to minimize the sparsity, we set a threshold for word frequency, and set any words lower than that threshold to unknown labels. For threshold, we tried 1, 2, 5.

TH	Seen	Unseen	TH	Seen	Unseen
2	14,471	23,033	5	8657	28,847

Table 2: TH: threshold for unknown words filtering; seen:words occur more than threshold times; unseen: filtered words

3 Handling Unknown Words (15pt)

To handle unknown words we used four techniques. First(Naive in Table 3), we replace any word with occurrence less than threshold with token $< unk >$. Second(Morpho in Table 3), we assigned number-like words (eg - one, twenty), digits, punctuation, all upper case and first word of sentence separate tag classes respectively. For other words, we use morphology classes to classify by their suffix or prefix into various tag classes. For eg, $'beautiful'$ with suffix $'ful'$ is $< JJLike >$ class. Third(Suffixgen in Table 3), in addition to last method, we also build suffix classes from training data. We look for suffixes of length 2, 3 and 4 in the training data and associate the suffix with specific tag if the suffix count for that tag exceeds a threshold for the given suffix size. We keep different suffix thresholds for different suffix sizes because suffix size of 2 is more common and hence needs a higher threshold. eg: In lovely: suffix of length 2-ly; 3-ely; 4-vely. Fourth(Thede in Table 3), per [1], open class POS tags are counted per word and then a dictionary of affix:POS tag counts is constructed. The longest affix per unknown word is identified. A tag is sampled from the POS distribution that was mapped to that affix. For eg:, if the word 'address' appears 100 times as a noun and 15 times as a verb in our training set, we first save the counts in a dictionary as {"address" : {"NP":100, "VB": 15}}. We then update our affix:POS tag counts by adding them for the word to the following affixes: 's', 'ss', 'ess', 'ress', and 'dress'. To tag the unknown word 'press' we sample a tag from the POS distribution of "ress" - the longest suffix match. Morpho works the best because, other methods seem contributing to more noise in unknown word selection.

4 Smoothing (10pt)

Our HMM probabilistic model used distribution of words, tags and ngrams of tags to estimate the maximum likelihood. However data sparsity is a major problem. We might have insufficient training data, that many words or ngrams in dev data doesn't have counts, and the model would give probability 0 to unseen words, leading to a wrong path or runtime error. In order to achieve smoothing, we should first avoid changing original distribution too much and make the most use of the corpus information; second, we should fill the gaps as much as needed. Following are the smoothing methods we explored, examples are all in bigram model:

1. Add-K smoothing In order to calculate transition, for all word bigram pair, add a λ to all the counts to numerator, and add a λ*total counts of tag to the denominator. However this gives all unseen words or tags receive same probabilities, which is not optimal.

2. Linear/Deleted Interpolation $P_(y_i - 1, y_i) = \lambda * P_{ML}(y_i - 1, y_i) + (1 - \lambda) * P_{ML}(y_i)$ where $0 < \lambda < 1$ Higher n-grams are sensitive to more context but the counts are very sparse, so we combine tighter lower grams which has limited context information. Deleted interpolation is similar to leave-one-out cross-validation in that each trigram is successively deleted from the training corpus and the λs are chosen to maximize the likelihood of the rest of the corpus. In our experiment, it boosts the performance for 0.6% compare to using add-k smoothing. The optimal weight for trigram model is [0.125, 0.394, 0.481].

3. Kneser-Ney. As described in [2, 3] we implement modified Kneser-Ney smoothing for bigram transitions and emissions per $P_{KN} = \frac{max(c(w_{i-1}, w_i) - D, 0)}{\sum_{w'} c(w_{i-1})} + \lambda_{w_{i-1}} P_{KN}(w_i)$. The main idea of KN smoothing is to consider lower order n-gram continuations instead of raw probabilities. It elegantly lowers predicted probabilities of common words that only appear after few distinct words. We lose some mass from the discounted probabilities, but it is redistributed by the λ term to increase overall results.

t	model	n	smooth.	unk.	param	dev F1	unk F1	test F1
2	greedy	2	add-k	naive	k=0.00001	0.9295	0.351	n/a
2	greedy	2	add-k	morpho	k=0.00001	0.9415	0.582	n/a
2	beam(b=4)	2	add-k	morpho	k=0.00001	0.9481	0.584	n/a
2	beam(b=4)	2	add-k	suffixgen	k=0.00001	0.9511	0.591	n/a
2	beam(b=5)	2	add-k	suffixgen	k=0.00001	0.9527	0.593	n/a
2	beam(b=6)	2	add-k	suffixgen	k=0.00001	0.9525	0.589	0.9520
2	viterbi	2	add-k	morpho	k=0.00001	0.9506	0.593	n/a
2	viterbi	2	add-k	morpho	k=0.000001	0.9508	0.597	**0.9556**
2	f/w-b/w vtrbi	2	add-k	morpho	k=0.000001	0.9432	0.574	n/a
2	viterbi	2	add-k	suffixgen	k=0.000001	0.9540	0.587	0.9533
5	viterbi	2	add-k	None	k=0.001	0.948	0.353	n/a
5	viterbi	2	add-k	morpho	k= 0.001	0.952	0.579	n/a
2	viterbi	2	add-k	morpho	k=0.001	0.9577	0.594	n/a
2	viterbi	2	add-k	morpho	k=0.0001	0.9579	0.595	n/a
2	viterbi	2	interpol	morpho	lam=0.6	0.959	0.598	n/a
1	viterbi	2	interpol	morpho	lam = 0.6	**0.9618**	0.595	n/a
1	viterbi	2	add-k	morpho	k=0.0001	0.9618	0.597	n/a
1	viterbi	3	interpol	morpho	[0.125, 0.394, 0.481]	0.957	0.593	n/a
1	viterbi	3	None	morpho	None	**0.9687**	0.653	**0.95825**
1	viterbi	3	add-k.	morpho	k=0.0001	0.9655	0.604	n/a
1	viterbi	3	Add-K	Thede	.01	0.9384	.606	**0.94305**
1	viterbi	2	Add-K	Thede	.01	0.9380	.598	n/a
1	viterbi	2	MD-KN	Thede	NA	0.9381	.615	n/a
2	Greedy	4	Add-k	morpho	NA	0.8351	.342	n/a

Table 3: t:threshold for rare word frequency; n: n-gram; smooth.: smoothing method; unk.: unknown handling method; param: corresponding smoothing parameters, unk-F1: unknown F1 scores

5 Implementation Details (5pt)

To implement the HMM Model, we computed the logs of emissions and transitions and stored them as a dictionary instead of 2-D array due to the sparse nature of the data. For HMM model, the word emission probability only depends on its own tag and independent from the neighbor words or tags; the transition probability for a tag only depends on previous (2 tags trigram, 1 for bigram) rather than entire sequence. We implemented 2 optimizations allowing the trigram tagger to run in 21 minutes on the dev set. Firstly, we skip impossible transitions with the trellis array architecture by floor-dividing the index of the current trellis node by the number of tags and then iterating by 46 to find the next previous state to check. Secondly, we exclude nodes in which the unsmoothed bigram emissions of the second POS to a word are zero. For four-gram, we calculate emissions and transitions on-demand to save space, and skip trellis nodes for which unsmoothed trigram/bigram emissions and transitions are 0.

6 Experiments and Results

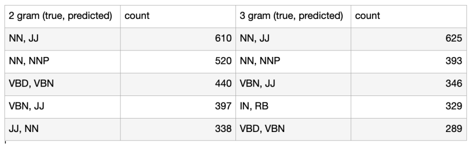
Test Results (3pt) Refer to Table 3 for test results. Our best test result(f1=0.958) uses viterbi trigram, with threshold 1, no smoothing, and morphology to handle unknown words.

Smoothing (5pt) We used Add-K, interpolation(deleted interpolation) and KN smoothing. In table 3, our result shows that, for bigram, interpolation(λ=0.6) performs better than add-K(k=0.00001), and KN interpolation out performed the others. For trigram, we tried deleted interpolation, add-k and non-smoothing, interestingly non-smoothing performs better than deleted interpolation([0.125, 0.394, 0.481]) and add-k. The reason might be trigram already has context information therefore no need to be weighted. We also apply trigram with KN smoothing on small batches. Overall, we conclude KN smoothing outperforms other smoothing methods.

Algorithm	Avg Sensitivity for all classes	Freqency of optimal solution
Greedy(b=1)	0.8471	0.0541
Beam(b=2)	0.8636	0.1406
Beam(b=3)	0.8766	0.2316
Beam(b=4)	0.8744	0.2683
Beam(b=5)	0.8831	0.3073
Beam(b=6)	0.8793	0.3160

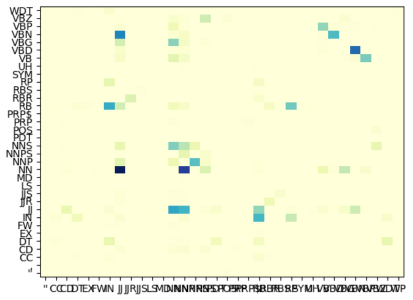
Table 4

Figure 1: Confusion Mat subset



2 gram (true, predicted)	count	3 gram (true, predicted)	count
NN, JJ	610	NN, JJ	625
NN, NNP	520	NN, NNP	393
VBD, VBN	440	VBN, JJ	346
VBN, JJ	397	IN, RB	329
JJ, NN	338	VBD, VBN	289

Figure 2: Heatmap of confusion matrix(trigram)



Bi-gram vs. Tri-gram vs. Four-gram(5pt) We see that the trigram model performed the best. Although it performs better, it takes almost 20 times more time to get the results than bigram. Hence we heavily experimented with various parameters in bigram as is evident from table 3. Bigram was used for fine-tuning various parameters. Trigram model performance is expected to be better due to the additional context present. Four-gram model gives lower score than trigram with exponential time cost which may be due overfitting of context.

Greedy vs. Viterbi vs. Beam (10pt) In Table 3, viterbi gives the best results compared to other techniques. This is expected because the viterbi looks for optimal paths by considering all possible options although at the cost of speed. Still we see that on using dictionary, it still runs quite fast for bigram. On seeing the success of viterbi the gap in prediction of words such as adjectives which come before noun/verb, we tried implementing viterbi in both forward and backward direction and choosing the tag with the higher score of both forward and backward trellis.Unfortunately this decreased our performance for bi-gram and hence we abandoned the idea. Viterbi algorithm performs the best with the trigram which is no surprise as that is the state of the art POS tagging available.The greedy decoder gets optimal solution in 5.41% of the sentences in training data. The average sensitivities over all classes and frequency of optimal solution is listed in the Table 4.

7 Analysis

Error Analysis (7pt) We saw our two best dev-performance model(bigram f1=96.2, trigram f1 = 96.878), see Fig1 for top 5 most frequent fail cases. We saw that both models confused noun and adjective. One reason is that, in our morphology classes, adj classes has more suffix hence many noun ended with adj suffix will be classified into adj, e.g'satellite-assembly' was tagged as JJ becasue 'ly' is a adj suffix. 'NN' and 'NNP' are also easily confused by both models since they are very similar, one way we can improve is by capitalized. Also, we saw that, trigram performs better in tagging 'VBD' and 'VBN' than bigram, we think it's because trigram has more context information, hence would tag past tense better.

Confusion Matrix (5pt) Refer to fig1, 2. Among verb classes, (VBN and VBD), (VBP and VB) are easily confused, (VBN and VBP) are never confused. For example, 'cut' with no form change in past tense was easily confused as past tense or VBN when it actually is VB. Verb and noun, verb and adjective are easily confused. For eg:, 'move' can be both verb and noun; 'split' can be both VB, VBD and NN. Other than that, RB and In, RB and RP, are very easily confused. These kind of tasks are hard even for humans.

8 Conclusion (3pt)

We found the POS Tagging on WSJ dataset to yield a best dev F1 score of 0.9687 and best test F1 score of 0.9582. Also, we found the lower threshold, the higher the F1 score.We also discovered KN smoothing > interpolation > add-K > no smoothing for most cases in this task.

A Appendix: Kneser-Ney Formulation

$$P_{KN}(w_i) = \frac{|\{w' : 0 < c(w', w_i)\}|}{\{(w', w'') : 0 < c(w', w'')\}|}$$

$$\lambda_{w_{i-1}} = \frac{D}{\sum_{w'} c(w_{i-1})} |\{w' : 0 < c(w_{i-1}, w')\}|$$

and D is a function of $c_b = c(w_{i-1}, w_i)$ where

$$D(c_b) = \begin{cases} 0 & c_b = 0 \\ 1 - 2Y\frac{n2}{n1} & c_b = 1 \\ 2 - 3Y\frac{n3}{n2} & c_b = 2 \\ 3 - 4Y\frac{n4}{n3} & c_b \geq 3 \end{cases}$$

in which Y is $\frac{n_1}{n_1+2n_2}$ and n_x is the number of bigrams with count of x in our dataset.

References

[1] S. M. Thede, "Predicting part-of-speech information about unknown words using statistical methods," in *COLING-ACL*, 1998.

[2] D. Jurafsky, "Lecture 19 — kneser ney smoothing — [nlp —— dan jurafsky —— stanford university]."

[3] S. F. Chen and J. Goodman, "An empirical study of smoothing techniques for language modeling," *Computer Speech Language*, vol. 13, no. 4, pp. 359–394, 1999.