# CIS 5300: Assignment 2
# POS Tagging using HMMs

Rishi Ghia (ghiar@seas.upenn.edu)
Santnam Bakshi (santnam@seas.upenn.edu)

## 1 Introduction (5pt)

For the purpose of this homework project, we have built a POS Tagger that has the capability to take an input sequence of words, and predict POS tags for each word. This was done through the use of various decoding techniques such as using a Greedy Algorithm, Beam Search Algorithm and finally, the Viterbi Algorithm. Both bigram and trigram based models were evaluated as well. We performed smoothing to deal with cases where a bigram or trigram was not seen in the training dataset, and also handled unknown words through a combination of various techniques to help improve the accuracy of the model.

The dataset used contains 1 Million words of text from the Wall Street Journal. The sentences in the dataset are written out word-by-word in a flat, column format in the *train_x.csv* file, where individual documents are separated by -DOCSTART- tokens. Each of these words has a corresponding POS tag, located in the $train_y.csv$ file.

For smoothing, we used the methods of Add-K smoothing, Witten-Bell smoothing, and Linear Interpolation. To handle unknown words, we focused on identifying the tags using the prefix, suffix and the capitalization (or lack thereof) of the first letter of the word.

Through these techniques combined with our Trigram Viterbi model, we achieved an F1 score of 95.51692. And an unknown accuracy of 70.49%

## 2 Data (5pt)

In our endeavor to build the POS tagger, we utilized the entire Wall Street Journal dataset tagged with the Penn Treebank. We used the given split of the development dataset and test set as is and did not perform splits on the data of our own.

### 2.1 Dataset Statistics

We did not make any changes to the casing of the data, and we did not perform any tokenization on the text either.

The training data-set has about 696,000 words split over 1387 sequences. The development data-set which was used to validate our model has 243,000 words split over 465 sequences. The test data-set which we finally tested our model on has 236,000 words split over 462 sequences. The sequences started with DOCSTART tag, and range from as short as 28 words to over 3000 words long.

## 3 Handling Unknown Words (15pt)

We started with a baseline approach of assigning each unknown word the tag of a Noun. This gave us a rather poor unknown word accuracy in the neighborhood of 15%. After that we implemented a more sophisticated approach to dealing with unknown words. To setup, we created a dictionary that stored the prefixes and suffixes of each word in the training dataset, and mapped it to the most common tag for each particular prefix and suffix in the training dataset.

When assigning the tag of an unknown word in our algorithm, we first checked if the first letter of the word was capitalised. If this was satisfied we assigned it the tag of a proper noun. If this condition was not satisfied, we tested to see if the suffix of the tag was in our dictionary of suffixes and assigned it to the tag of the most popular suffix. If the suffix was not recognised, we checked the prefix of the word and assigned it to the tag of the most popular prefix. If the prefix was not satisfied, then we assigned it a default tag of a noun. We tuned both the size of the prefix and the size of the suffix and the order in which to use these three ways of assigning the tag. We obtained the highest unknown word accuracy for the above stated order, a prefix size of the first 2 letters of a word and a suffix size of the last two letters of

a word.

For instance, if the unknown word is "unavailability". We check the first letter and see that it is not capitalised, we then check the suffix, which is 'ity'. We then check the suffix dictionary for the most common tag for words ending with 'ity' in the training dataset, and assign that as the tag. If 'ity' is not found in the suffix dictionary, we then check the prefix dictionary for the most common tag for words starting with 'un' in the training dataset, and assign that as the tag. If 'un' is not found in the prefix dictionary, we assign the tag as noun.

We then obtained the transition probability using this tag assigment, and we set the emission probability as the probability of the occurrence of the tag in the training dataset, to assign it a low probability value. This does not affect our final resulting sequence because we do not use the emission probability to determine the tag in the case of unknown word, and the tag is already determined.

This approach makes sense, because a capitalised word has a high probability of being a proper noun and the suffix and prefix of a word is reasonably indicative of the tag of a word. The default tag being a noun is reasonable because that is the most common tag for an unknown word, without considering any information of the word. Our approach however is a little simplified and we haven't made the effort to consider prefixes and suffixes of different lengths, or to consider cases where the first word of a sentence is capitalised and need not be a proper noun. There are also cases where the word may have a more complex morphological structure and the prefix and suffix may not predict the correct tag. We stuck to this approach because of its simple implementation and above average results. We obtained an unknown token accuracy of 70.48% with this simple approach.

## 4 Smoothing (10pt)

In our exploration of smoothing techniques, we adopted three distinct methods to tackle the challenge of zero probabilities for unseen events. Given below is a brief overview:

**Add-k Smoothing:**
This method involves adding a constant $k$ to each count, thus ensuring non-zero probabilities for unseen events. The formula we used is:

$$P(event) = \frac{count(event) + k}{totalcount + k \times V}$$

where k is a constant and a hyper-parameter that we tuned. And V is the size of the Vocabulary. **Witten-Bell Smoothing:**
Primarily beneficial for bigram models, this technique reallocates probability mass from observed to unobserved events based on unique occurrences.

**Linear Interpolation:**
Here, we combined the probabilities of n-grams and their lower order counterparts to estimate probabilities for unseen n-grams. For trigrams:
$\mathrm{P}(\mathrm{w}_3|w_1, w_2) = \lambda_3 P(w_3|w_1, w_2)$
$+ \lambda_2 P(w_3|w_2) + \lambda_1 P(w_3)$

**Why Add-K Stood Out:**
Through our evaluations, we found that Add-K smoothing was particularly effective with the WSJ dataset. The dataset's vast vocabulary and infrequent n-grams made the uniform adjustments of Add-K smoothing beneficial. Additionally, given WSJ's structured nature and high frequency of certain n-grams, Add-K's minimalistic approach ensured more consistent predictions.

**Trade-offs in Smoothing Techniques:**
While smoothing enhanced our model, there were inherent trade-offs:

- **Add-k Smoothing**: The uniformity assumption might misrepresent the likelihood of some rare events. Additionally, redistributing probability mass can affect the prominence of frequent n-grams.
- **Witten-Bell Smoothing**: Basing adjustments on unique observed events might not always be optimal, and the assumption of uniformity for unseen bigrams may sometimes be off-mark.
- **Linear Interpolation**: This method can sometimes overshadow higher-order n-grams, and the need to tune interpolation weights added complexity.

In conclusion, our journey through these smoothing techniques offered insights into their strengths and limitations. Choosing the right method required an understanding of both the dataset's nuances and the inherent properties of each technique.

## 5 Implementation Details (5pt)

Regarding the 'get_unigrams', 'get_bigrams', and 'get_trigrams' functions, there isn't much to elaborate on beyond what was defined in the assign-

ment itself. We integrated our smoothing methods within these functions prior to computing the final probabilities for the unigrams, bigrams, and trigrams. Within the 'init' function, we established all our count dictionaries and also initialized our hyperparameters, such as $k$, the three lambdas for smoothing, and the $k$ value taken as an argument for beam $k$.

In 'get_emissions', we iterate over all combinations of words and tags to calculate their probabilities, storing the results in a dictionary.

Throughout this assignment, we prioritized the use of dictionaries and sets wherever we needed to retrieve a specific value from a large collection. This ensured that the lookup time was minimized, especially since numerous lookups are performed within loops. This helped us bring down the inference time for Viterbi Trigrams from 90 minutes to 6 minutes, constituting a 15x improvement, allowing for a lot more hyperparameter tuning and other improvements.

The 'train' function is where all the n-gram initializing functions are invoked. We also constructed two dictionaries, mapping the bigrams to their indices and vice versa. This aids us with the trigrams in the Viterbi algorithm since we employ bigrams in the state lattice there. Additionally, this is where we developed the prefix and suffix to tag dictionaries. Using a probability-based approach, these dictionaries map a word's prefix or suffix to its most probable tag.

The inference function is where we call our models, greedy, beam or Viterbi.

**Greedy:** In our greedy method, we utilize a greedy decoding strategy to tag a given word sequence based on either bigram or trigram probabilities. For known words, we calculate transition probabilities based on previous tags and select the tag with the highest product of transition and emission probabilities. For unknown words, we employ heuristics: considering it a proper noun if its initial letter is capitalized or using suffix and prefix mappings to guess the tag. If these heuristics don't provide a tag, we default to labeling the word as a noun. We maintained efficiency by ensuring that every operation was performed outside a loop if possible to minimize the amount of computation.

**Beam:** In our beam method, we employ a beam search approach for part-of-speech tagging, considering either bigram or trigram probabilities based on the kgram attribute. For each word in a sequence, if it's known, we maintain a list of the top k most probable tag sequences. We expand each sequence by one tag and compute its probability, but only keep the top k sequences for the next iteration. For unknown words, we apply heuristics similar to the greedy method. Towards the end, we return the highest probability sequence. Using beam search, we strike a balance between the exhaustive search, which considers all possible sequences, and the greedy approach, which only looks at the most probable tag for each word, making our method more computationally efficient while maintaining reasonable accuracy. Naturally, k was taken as a hyperparameter, and k = 3 gave us the best results. Taking k to be higher or lower than 3 had adverse effects on the F1 Score.

**Viterbi:** In our 'viterbi' method, we harness the Viterbi algorithm for part-of-speech tagging, determining the most probable sequence of tags for a given word sequence. If we're in the bigram scenario, for each word, we compute the likelihood of every potential tag based on the previous tag and keep track of the highest probabilities and their paths. For trigrams, the process considers pairs of preceding tags. For known words, we use transition (from bigrams or trigrams) and emission probabilities. Unknown words are managed using heuristics, similar to previous methods. After processing the sequence, we backtrack using stored paths to construct the optimal tag sequence. This dynamic programming approach ensures we find the best tag sequence by efficiently exploring possible paths, balancing computational intensity with tagging accuracy.

## 6 Experiments and Results

**Test Results (3pt)**

| Sr. No. | Model | N-Gram | Smoothing Technique | F1-Score (test_y) |
|---|---|---|---|---|
| 1 | Viterbi | Trigrams | Add-K | **95.51692** |
| 2 | | | Linear Interpolation | 95.16374 |
| 3 | | Bigrams | Add-K | 95.51507 |
| 4 | | | Witten-Bell | 95.49733 |
| 5 | Beam (k=3) | Trigrams | Add-K | 95.38521 |
| 6 | | | Linear Interpolation | 94.99362 |
| 7 | Beam (k=2) | | Add-K | 95.26746 |
| 8 | | | Linear Interpolation | 94.60387 |
| 9 | Beam (k=3) | Bigrams | Add-K | 95.35552 |
| 10 | | | Witten-Bell | 95.32671 |
| 11 | Beam (k=2) | | Add-K | 95.11244 |
| 12 | | | Witten-Bell | 95.08446 |
| 13 | Greedy | Trigrams | Add-K | 94.80705 |
| 14 | | | Linear Interpolation | 94.02681 |
| 15 | | Bigrams | Add-K | 94.67206 |
| 16 | | | Witten-Bell | 94.67305 |

As seen above, the best model that we created has the following specifications:
**Model Name:** Viterbi
**Smoothing Technique:** Add-K Smoothing (k = 0.1)
**N-Gram representation:** Trigrams
**Unknown Word Technique:** Prefix, Suffix Trees, First Letter Capitalization

**Smoothing (5pt)** We observe that Add-K is the most effective smoothing technique for Viterbi and Beam Search. Meanwhile, Witten-Bell is better in the case of Greedy with Bigrams. Add-K was found to perform significantly better than Linear Interpolation in the case of trigrams. An analysis of why this is the case can be found in the section above where our smoothing methods have been described.

**Bi-gram vs. Tri-gram (5pt)** Trigrams with the same smoothing technique, outperform their Bigram counterparts. Yet, Trigrams with Linear Interpolation perform worse than Bigrams with Witten Bell. These results are consistent across models.

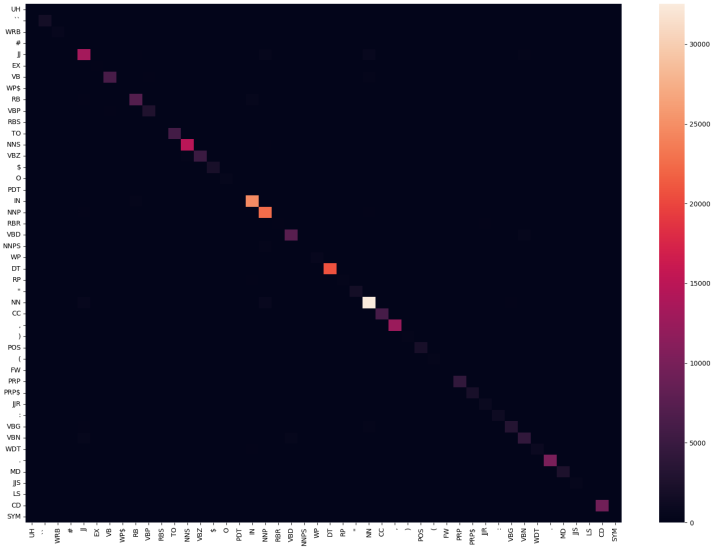
**Greedy vs. Viterbi vs. Beam (10pt)** We find that when keeping the smoothing technique, N-Gram model representation and other hyperparameters constant, Viterbi significantly outperforms all other models. This can be attributed to the Dynamic Programming techniques that it is built upon, that allow it to perform an exhaustive search across the domain.

Beam Search with k = 3 performs very well, and outperforms Beam with k = 2. Both of these outperform the Greedy Algorithm as expected.

## 7 Analysis

**Error Analysis (9pt)** There were certain tags that we noticed errors for quite often. There were certain instances where the Noun tag was predicted as an Adjective tag. The tenses of certain verbs were also predicted incorrectly. Additionally, the singular and plural nature of some nouns were predicted incorrectly. We noticed that this was mostly in the case of unknown words, and can be attributed to the relative simplicity of the method we have used to deal with unknown words (where we achieved an accuracy of about 70%).

**Confusion Matrix (5pt)** As seen here, the highest correlation is seen in nouns, prepositions, Proper Nouns, and Determiners. This is understandable considering that the tags which occur the most in the dataset will also have higher accuracy thus giving us a better correlation.




## 8 Conclusion (3pt)

During this assignment, we delved deep into the intricacies of part-of-speech tagging, employing various algorithms and techniques to enhance accuracy. By leveraging n-grams, the Viterbi algorithm, and different heuristics, we built a robust tagging system. The use of dictionaries and sets ensured optimal time complexity during lookups, highlighting the importance of data structure selection in natural language processing tasks. This exercise not only solidified our understanding of POS tagging but also underscored the significance of balancing algorithmic complexity with computational efficiency in real-world applications.