# Ghoti Chinese Word Segmenter

## Algorithm outline

The final version uses Jelinek-Mercer smoothing to combine bigram and unigram probabilities together with a simple length-based estimation function for unseen words. The core maximizer is still Dynamic Programming, based on the given pseudocode but with some minor efficiency improvements so that we can afford longer unseen words. Finally, a heuristic tweak is added to deal with the "full-width" (Unicode) digits that are very common in the data.

All combined, this version achieves a F1-score of 89.94 locally and 93.55 on the leaderboard with rather clean code and very few hard-coded parameters.

## Probability model and estimation function

Effectively, we are cutting a sentence into $n$ words ($W_{1..n}$) to maximize the following probability (assume $W_0$ is some special token for a line start):

$$P(W_{1..n}) = \prod_{k=1}^{n} P(W_k|W_{k-1})$$

Where $P(W_k|W_{k-1})$ is the probability of seeing word $W_k$ given $W_{k-1}$ immediately in front, which is a bigram model. Now to estimate this function, we look at two parts, bigram-based estimation $P2$ and unigram-based estimation $P1$, and blend them together linearly (Jelinek-Mercer smoothing):

$$P_{est}(W_k|W_{k-1}) = \lambda P2(W_k|W_{k-1}) + (1-\lambda)P1(W_k)$$

$$P2(W_k|W_{k-1}) = \begin{cases} count2w(W_{k-1}, W_k)/count1w(W_{k-1}), & \text{if this pair of words appears in the bigram count} \\ 0, & \text{otherwise} \end{cases}$$

$$P1(W_k) = \begin{cases} count1w(W_k)/\sum_i count1w(W_i), & \text{if the word appears in the unigram count} \\ \{\min_i[count1w(W_i)]/\sum_i count1w(W_i)\}^{len(W_k)}, & \text{otherwise (*)} \end{cases}$$

Some notes:

- Formula (*) strictly speaking is not a probability distribution, since the sum of all (infinite number of) unseen words would well exceed 1. Actually, even if $P1$ is a real probability distribution function, the linear blending $P_{est}$ is also by no means a probability. The two parts being added up, $P1$ and $P2$, have totally different meanings and scales and we are sort of tweaking the parameter $\lambda$ to blend the two different scales nicely. In a word, we could only say $P_{est}$ is related to the probability, especially in terms of the relative magnitude of the values.
- Formula (*) is based on the idea of severely "punishing" long unknown words. An unknown word of one character is considered similar to the rarest word in the unigram count. Longer unknown words fall off exponentially.

Parameters in practice (from experiments):

- $\lambda = 0.9$ gives the best result.
- The rarest word has a count of 1 in our unigram data.

Last but not least, The maximal length of a single word we may try is the maximal length of the words in the unigram, which could be unknown (and estimated by Formula (*)).

In terms of result (F1 score), we first added Formula (*) to estimate unknown words which boosted the local score from the baseline 80 to 86; and then the bigram and the Jelinek-Mercer smoothing boosted the local score to 90+.

## Adjustment on DP routine

The original DP pseudocode stores the "entries" (the last word we segment in the computed prefix of the sentence) in a heap as a speed improvement to avoid unnecessary enumeration. At the core, it is still doing the classic 1-dimensional DP:

$$F[i] = \max_{j=1}^{maxlen} F[i-j] + P1(S[i-j..i])$$

Where $S[i..j]$ is the substring from position $i$ to $j$ in the input sentence, and $P1$ is the unigram probability given in the baseline algorithm. Note that we transform the product into summation by logarithm.

However, in order to maximize the aforementioned probability model, we need a 2-dimensional DP:

$$F[i,j] = \max_{k=1}^{maxlen} F[j-1, j-k] + P_{est}(S[i..j], S[j-k..j-1])$$

Where $i$ is still the ending position, but now $j$ is the previous ending position, i.e., $S[i..j]$ is the last word, since we need to know the last word in order to compute the bigram probability.

The given pseudocode can actually compute the 2D DP with some minor modifications, thanks to the entry-based iteration. First, we simplify the definition of entry to represent exactly our states in the 2D DP: entry is now just the ending position and the last word. All other information goes into external dictionaries. Most importantly, the keys of the DP chart are no longer simply positions, but entries. And now the DP would work. By doing so, we also greatly reduce the size of each entry, which turns out to benefit the performance.

There is one more minor, but important, "side-effect" of the modification on entry definition: the original pseudocode would possibly push two entries that are at the same position but with different probabilities, as the probability is in the entry tuple and the two entries would thus be considered different. This is unnecessary work, because the one with the lower probability would never win. By taking the probability out from the entry into the chart, we will no longer push such duplicate entries. This really impacts the performance when we accept long (say up to 15 characters) unknown words (and thus much more potential duplicates). Originally, it would take 20 minutes to finish; after this modification, it only takes less than 30 seconds.

## Heuristics for full-width digits

There are a lot of "full-width" digits (i.e. characters that appear similar to Arabic numerals but are not within ASCII range), as well as the full-width version of the decimal point. They could be rather arbitrary and severely affect the result. We normalize these digits into ASCII digits and encourage them to be grouped together. This heuristic gives about 1~2% increase.

## Other experimented approaches

Our team members also played around with many different methods, including those mentioned in the class such as Good-Turing smoothing, additive smoothing, and Katz back-off. All of them either performed very

badly on this particular task and data set, or gave ignorable improvement. See the individual reports and the subdirectories for details.