Homework 3
Advanced Natural Language Processing

Felipe del Canto

October, 2021

1. Part 1: Hidden Markov Models

In order to create the HMM object, I first initialize the transition probability matrix A and the emission probability matrix B as follows. First, generate a $n \times m$ matrix of random numbers taken from a uniform distribution. For A, $n = m =$ num_states and for B, $n =$ num_states and $m =$ num_words. Second, divide each row by the sum of the row. This way, the rows of A and B sum up to 1. Additionally, in order to initialize the initial distribution vector, π, I randomly sample a num_states-dimensional vector from a uniform distribution and normalize it to sum up to 1 by dividing by the sum of its elements.

Now, to code the forward method I follow the forward algorithm. Since this is being made in log-space, the formulas must be adjusted accordingly. Specifically, if $\boldsymbol{O}_t$ is the t-th element of the sentence, the forward algorithm is given by:

$$\begin{aligned}\alpha(1,i) &= \pi_i B_{i,o_1}\\ \alpha(t,i) &= B_{i,\boldsymbol{O}_t}\sum_j \alpha(t-1,j)A_{ji}\end{aligned}$$

Then the algorithm in log-space is

$$\begin{aligned}\alpha(1,i) &= \log(\pi_i) + \log(B_{i,\boldsymbol{O}_t})\\ \alpha(t,i) &= \log(B_{i,\boldsymbol{O}_t})\\ &\quad + \text{logsumexp}\left(\sum_j \alpha(t-1,j) + \log(A_{ji})\right)\end{aligned}$$

Similarly, the backward algorithm must also be transformed when coding the backward method. Initially, the algorithm is

$$\begin{aligned}\beta(T,i) &= 1\\ \beta(t,i) &= \sum_j A_{ij}B_{j,\boldsymbol{O}_{t+1}}\beta(t+1,j)\end{aligned}$$

and its transformed counterpart in log-space is

$$\begin{aligned}\beta(T,i) &= 0\\ \beta(t,i) &= \text{logsumexp}\Bigg(\sum_j \log A_{ij}\\ &\qquad + \log B_{j,\boldsymbol{O}_{t+1}} + \beta(t+1,j)\Bigg)\end{aligned}$$

Now, for the forward_backward method, the same kind of transformation has to be done. First, note that in log-space we have:

$$\log P(\boldsymbol{O}) = \text{logsumexp}\left(\sum_i \alpha(T,i)\right)$$

With this, we have that

$$\log \gamma(t,i) = \alpha(t,i) + \beta(t,i) - \log P(\boldsymbol{O})$$

And finally, for ξ,

$$\begin{aligned}\log \xi_t(i.j) &= \alpha(t,i) + \log A_{ij} + \log B_{j,\boldsymbol{O}_{t+1}}\\ &\quad + \beta_{t+1}(j) - \log P(\boldsymbol{O})\end{aligned}$$

And thus,

$$\xi_t(i,j) = \exp\left(\log \xi_t\right)$$

Note that, since ξ depends on t, i and j, then ξ (and $\log \xi$) is stored as a 3-axis Tensor, where the first axis corresponds to t and for each t, $\xi[t,:,:]$ is a num_states × num_states matrix.

Finally, to code the learn_unsupervised, after initializing the necessary quantities to zero, the computations follow the usual formulas of the algorithm:

$$\begin{aligned}\hat{E}(s_i \to s_*) &= \sum_{t=1}^{T-1}\gamma(t,i)\\ \hat{E}(s_i \to s_j) &= \sum_{t=1}^{T-1}\xi_t(i,j)\\ \hat{E}(s_i, w_k) &= \sum_{\substack{t=1\\ \boldsymbol{O}_t = w_k}}^{T}\gamma(t,i)\end{aligned}$$

In particular, to compute the last expression, I construct a matrix mask of size num_words × len(review) such that entry (k,t) is 1 if $w_k = \boldsymbol{O}_t$ and zero otherwise. This is done by first creating a matrix (check_word_matrix) of the same size of the mask in which each column is the range from 0 to num_words, representing the symbols for each word. Then, the mask is just the matrix review == check_word_matrix, which checks which word is in which time. For example, if $\boldsymbol{O}_t = w_k$, then when computing mask, the character in $\boldsymbol{O}_t$ will not coincide with the other $k-1$ symbols in the column t of check_word_matrix. Consequently, the entry (k,t) of mask will be 1 and the others will be 0. This construction implies that

$$\hat{E}(s_i, w_k) = \gamma(:,i)^T \texttt{mask}(k,:)^T$$

and the matrix $\hat{E}(s_i, w_k)$ can be computed by matrix multiplication. Observe that this operation does not cause underflow issues since mask is a binary matrix.

Now, for part (a), consider an HMM of 8 states. The idea is to pair these states in way that for a given state i, there exists only one state $j(i)$ such that the transitions $i \to j(i)$ and $j(i) \to i$ occur with probability 1. This way there are 4 possible sequence of states, each corresponding to one of the four sequences. Under this construction, the emission probabilities will be also binary. This is, in state i there is only one symbol $w(i)$ which has probability 1 of being emitted in that state. Analogously, this means that each state is also associated with a unique symbol. Finally, π will be such that the initial distribution of states would be uniform over the states associated with the first two possible symbols on the corpus: 0 and 1. Specifically, consider the following parameters for the HMM:

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\pi = \begin{bmatrix} 0.5 & 0 & 0.5 & 0 & 0.5 & 0 & 0.5 & 0 \end{bmatrix}$$

Observe that matrix A is a block diagonal matrix, with each block equal to

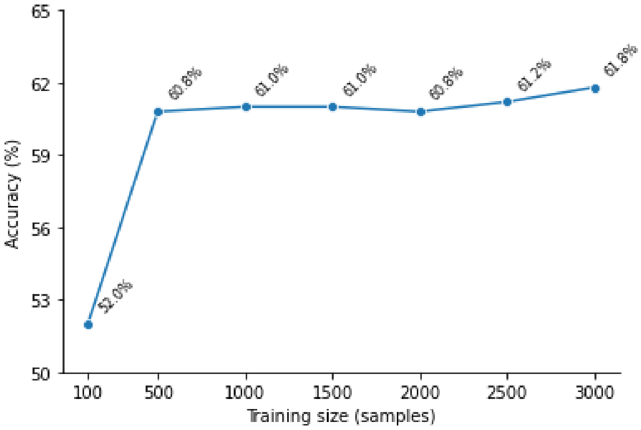
$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

This implies that pairs of states will cycle between each other. For example, states 1 and 2 will interchange forever in any sequence, since $A_{12} = 1$ and $A_{21} = 1$. In order to match each of these pairs to a given sequence, matrix B defines de emission probabilities accordingly. For example, for the states 1-2 cycle, we have that in state 1, the symbol 0 is emitted with probability 1, while for state 2, the symbol emitted is 2. This codes the sequence $[0, 2, \ldots]$ in the corpus. This works in the same fashion for the rest of the sequences. The vector π of initial states ensures the only states available at the beginning are the ones that emit the symbols 0 and 1.

For part (b), there does not seem to be a pattern with respect to what each state is decoding. I suggest that a bug in the code is preventing the HMM from learning correctly the parameters.

For part (c), the accuracies of the classifier when trained on different number of samples is presented in Figure 1. We can see that the performance when using only 100 examples is quite low, with a 52% of the test examples correctly labeled. However, the quality of the classifier improves sharply when increasing the training set. In fact, the performance from 500 examples onwards is stable and around 60%. The best performance is

Figure 1: HMM label prediction accuracy using a Logit model, for different training sizes. The HMM has 10 hidden states and the unsupervised algorithm for its parameters was trained for 10 iterations.



achieved when the complete training set is used, with an accuracy of 61.8%.

For part (d), note that given a v-sized vocabulary $\mathcal{V} = \{1, \ldots, v\}$, the bigram model is given by the following v^2 parameters:

$$\theta_{ij} = p(\boldsymbol{O}_t = i \,|\, \boldsymbol{O}_{t-1} = j), \quad i, j \in \mathcal{V}$$

To transform this into a HMM model, let the number of states be v. Consider the following parameters for the HMM model:

$$\begin{aligned} A_{ij} &= \theta_{ij} \\ B_{ik} &= 1\{i = k\} \end{aligned}$$

This is, the probability of transitioning from state i to state j is the probability given by the bigram model and the emission probabilities are 1 if the symbol coincides with the state. In other words, each state is associated with emitting a single symbol which is coded with the same number as the state (word i is emitted by state i). As we proved in HW1, the bigram (maximum likelihood) estimates are obtained by counting. For the HMM, since each state corresponds to a given word in the vocabulary, the learned parameters are also obtained by counting and the estimates are identical. Finally, π can be set to be the relative frequency of each of the words in the vocabulary that are in the first position.

2. Part 2: Trees

For part (a), first consider the following abbreviations:

- S: Sentence start.
- NP: Noun phrase.
- TNP: Terminal noun phrase.
- TV: Transitive Verb.
- IV: Intransitive Verb.
- D: Determiner.
- N: Noun.
- C: Conjunction.
- OC: Object Clause.
- CS: “Claused” sentence.

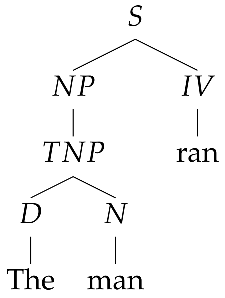
Now, consider the following rules:

$$\begin{aligned} S &\rightarrow NP \quad IV \\ NP &\rightarrow TNP \quad | \quad TNP \quad OC \\ TNP &\rightarrow D \quad N \\ OC &\rightarrow C \quad CS \\ CS &\rightarrow NP \quad TV \end{aligned}$$

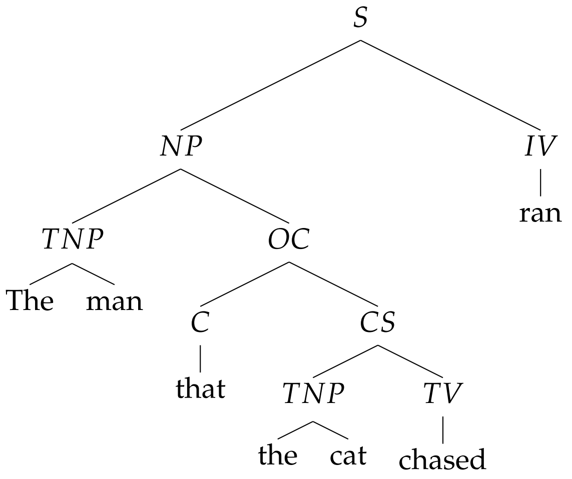
To show that these set of rules produce the sentences considered, consider the following steps. First, the sentence is created, consistent of two parts: NP and IV. At the same time, the NP part can lead to a TNP or a TNP *and* an OC. For example, if we have

$$S \rightarrow NP \quad IV, \qquad NP \rightarrow TNP$$

then the sentence is just TNP and IV. Since TNP is formed by a D and a N, then sentence (1), can be generated:



If, instead, $NP \rightarrow TNP \quad OC$, then the first object-oriented clause can be generated. Observe that OC is composed by the conjunction ("that") and a "claused" sentence. In this context, a CS is a NP (which in turn can lead to more OC) and a *transitive* verb (TV). For example, sentence (2) is generated as follows:



This construction continues if the NP in CS leads to another clause, allowing the rules to generate sentences (3) and (4).

Now, if we only wished to produce sentence with up to double clause nesting. Then there should be symbols that capture how many nestings are happening. In particular, consider the symbols NP_i, OC_i, and CS_i, which enumerate the current number of ocurrences of each symbol. Now consider the modified rules:

$$\begin{aligned} S &\rightarrow NP_1 \quad IV \\ NP_i &\rightarrow TNP \quad | \quad TNP \quad OC_i\,, \quad i = 1,2 \\ NP_3 &\rightarrow TNP \\ TNP &\rightarrow D \quad N \\ OC_i &\rightarrow C \quad CS_i \\ CS_i &\rightarrow NP_{i+1} \quad TV \end{aligned}$$

With this new set of rules, sentence (3) is still valid, as seen in Figure 2. This same image shows that sentence (4) is not generated anymore, since NP_3 does not allow a new OC to be included.

In order to complete Part A of this section, the first step was completing the `tree_dfs` function. Since this function is constructed recursively, when obtaining the span list and the label dictionary for each children, we only have to call the same function on the child. Next, to code the Bidirectional LSTM sentence encoder, the `nn.LSTM` module is used. Here, the input size of the LSTM is equal to the hidden size, since we are first embedding the input using the `nn.Embedding` module. The rest of the parameters are straightforward from the intended definition of the LSTM: the hidden size is equal to the input size (due to the reason outline above), the LSTM is set as bidirectional and a dropout option is included.

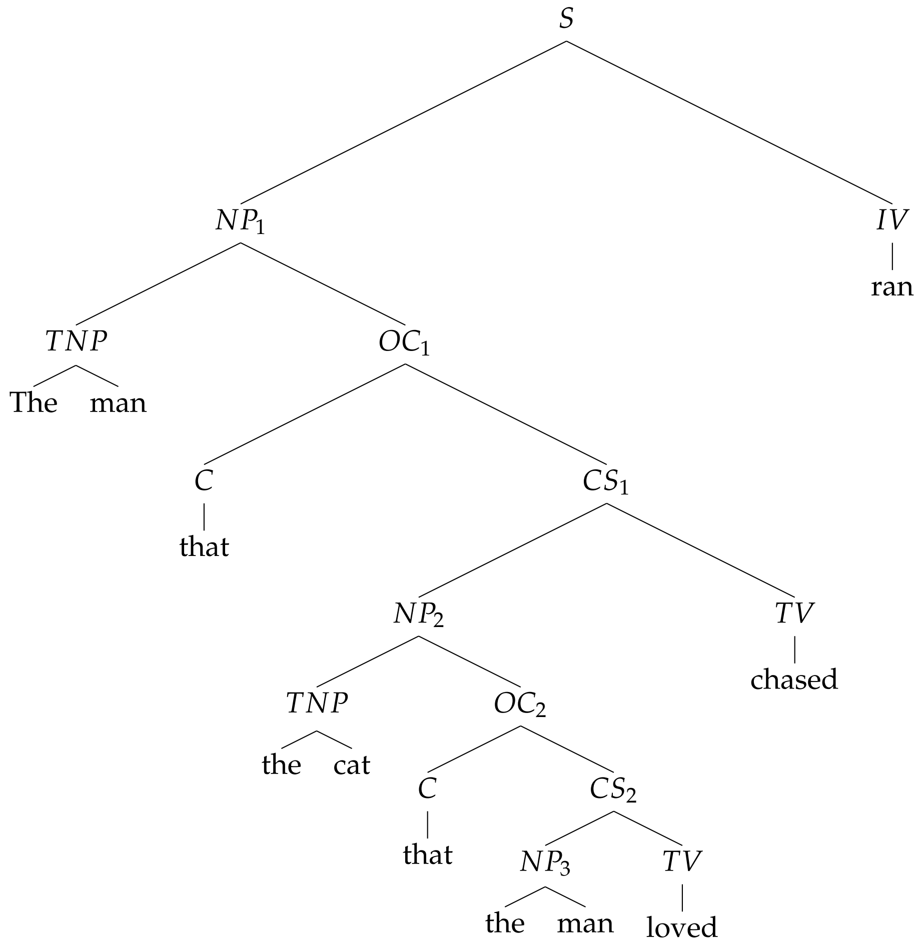
Next, to code the `get_span_embeddings` function, given a certain pair of span indices `(i,j)`, the embeddings are constructed by concatenating `h[i,:]` and `h[j-1,:]`, where `h` are the `sentence_length` × `hidden_size` hidden states obtained from the LSTM. In other words, the span starting at i and finishing at $j-1$ is embedded as the concatenation of the embedding of it first and last words.

Now, to construct the `Classifier` module, I first modified the `__init__` method to accept as dropout a dictionary indicating the dropout probabilities of the lstm and the word embeddings separately. For the `forward` method, the first step is to obtain the word embeddings from the LSTM and then apply a Dropout layer on them. The next step is to obtain the span embeddings using the `get_span_embeddings` function and finally obtaining the logits passing the span embeddings through a linear layer. Observe that under a logit model, the output of the linear layer are the (unnormalized) log probabilities of the span embeddings.

Finally, to code the training loop (which was refactored into a function), the first step is to define the loss function (in this case Cross Entropy), the optimizer (SGD) and the classifier. Next, for each epoch a complete pass over the training corpus is performed. For each sentence in the corpus, the current spans (including the sampled `None` spans) are extracted, and then split between indices (for the classifier) and labels (for computing loss). Finally, the predictions are obtained and the loss computed, making a backward pass and making a step in the optimizer. Before continuing to the next epoch, the same is made in evaluation mode for the evaluation corpus.

For part (a), the final scores obtained for dif-

Figure 2: Modified language generating sentence (3).



ferent parameters are presented in Table 1. As shown in the table, the metrics are generally stable between different parameterizations. However, slightly different improvements arise, especially in the metrics of interest. For example, the fourth row of the table suggests that having more layers improves the metrics in general. Compared to the second row, there were improvements in Recall, F1, Exact Match and Tree Match. These last two improved in 0.7 and 0.9 percentile points, respectively. Additionally, it is possible to observe that in general, increasing the embedding size of the spans also improves the representation. This improvement could be explained by an increase of the representation flexibility. Since what matters is finding good representations for spans, and for a given word many spans could be created, then having more dimensions to store information should improve the results. In this case, comparing rows 2 and 3, or rows 5 and 6 show an improvement in all metrics. Finally, regarding the dropout probability, rows 1 and 2, and rows 3 and 4 show that including Dropout layers in the LSTM improve the tree structure prediction, at the cost of precision. Overall, the best results were obtained with 3 layers, 400 hidden size and a Dropout layer in the LSTM with probability 0.25.

For part (b), the intuition behind using a bidirectional LSTM is as follows. The downstream taks in mind is correctly predicting the labels of the spans of the tree. Since these separate the sentences in blocks, it is important that the embedding for the initial word in the span to encode information on the following words, especially on the end of the span. An experiment ran with a unidirectional LSTM showed that indeed this intuition is correct. The Precision, Recall and F1 metrics decrease up to 0.585, 0.727, and 0.649, respectively. These values are far lower than the benchmark, shown in the second row of Table 1.

For part (c), there are several ways to improve the embeddings for each span. Currently, the algorithm only concatenates the LSTM hidden states (after applying dropout). This, with the idea of encoding the initial and final words in the span. However, if word embeddings are actually capturing some higher order semantic structure, then it could be possible to improve the span embedding by incorporating information about the inner words in the span. In other words, for span (i, j), instead of taking the hidden states for i and $j-1$, we could incorporate some transformation (e.g., mean) of the embeddings for words $k \in \{i+1, \ldots, j-2\}$. Following the same principle adopted for the original span embeddings method, including these hidden states also brings information about the inner structure of the span, potentially improving the tree decoding task. Moreover, following the argument in part (b) about the LSTM being bidirectional, the fact that the in-between words share information about each other (and, in particular, about the span limits) should be helpful in improving the performance in the downstream task.

To complete the code for Part B, the first step is to construct the dictionaries that iteratively store the best score of each span, the best split, and the best scoring label. To this end, we follow a variation of the CKY algorithm, and traverse the parsing chart from the bottom to the top using greedy decoding. In particular, given a span (i, j), the best score for

Table 1: Metrics of tree label prediction task for different parameters. All experiments were trained for 3 epochs, used 8997 examples, used a learning rate of 0.05 and applied a dropout layer ($p = 0.25$) on the classifier. In bold, the maximum of each metrics column. All values are rounded to the nearest thousandth.

LSTM parameters			Metrics					
Layers	Hidden size	Dropout p	Precision	Recall	F1	Exact Match	Well Form	Tree Match
2	200	0	**0.854**	0.943	0.896	0.414	0.547	0.478
2	200	0.25	0.848	0.945	0.894	0.416	0.542	0.479
2	400	0.25	0.850	0.948	0.897	0.419	0.544	0.483
3	200	0	0.852	0.946	0.896	0.424	**0.557**	0.491
3	200	0.25	0.848	0.949	0.896	0.423	0.541	0.488
3	400	0.25	0.851	**0.952**	**0.898**	**0.438**	0.548	**0.494**

that span is computed as

$$\begin{aligned}\texttt{best\_score}_{ij} = \max_{i+1\leqslant k\leqslant j} \big\{&\texttt{best\_score}_{ik} \\ &+ \texttt{best\_score}_{kj}\big\} \\ &+ \max_{\ell}\big\{\texttt{label\_scores}_{ij,\ell}\big\},\end{aligned} \tag{1}$$

where $\texttt{label\_scores}_{ij,\ell}$ is the score for label ℓ in span (i, j). To implement this recursion, we first iterate over the length of the span. In this manner, the spans of length 1 are the first checked (the ones at the bottom of the parse chart), then the spans of length 2, and so on. Observe that for the spans of length 1, there is no splitting point possible. Consequently, for these spans, (1) is just:

$$\texttt{best\_score}_{ij} = \max_{\ell}\big\{\texttt{label\_scores}_{ij,\ell}\big\}.$$

That is, for spans of length 1, the best score is the score of the most probable label. Consequently, the best label is the one that leads to the highest probability. For spans of length 2 or above, we use (1) to compute the best scores. Observe that the split is only inclusive in the second span. In order words, when the span (i, j) is split into (i, k), (k, j), the first consists of tokens up to $k - 1$, and the second span starts with token k. After computing the possible split scores, the best split is the one that maximizes the sum in the first term of (1), while the best label is the one that maximizes the second term.

After decoding the trees, the next step is to construct them. To this end, we need to complete the `dfs_build` function. This function receives a given span, the dictionary of best labels and splits and returns a node of the tree. Observe that since each node may have other nodes as children, the nature of this function is recursive. That is, for building the tree from span (i, j), it is necessary to build the tree for each pair of spans (i, k), (k, j) for $k \in \{i + 1, \ldots, j - 1\}$. These spans will, in turn, use the same function to build their respective nodes. Specifically, when $j = i + 1$, then no splits are needed and the function returns a Token node or a labeled node with one Token children. Otherwise, the span is split using the best split obtained before and the function is called recursively to obtain the nodes of splits (i, k) and (k, j).

For part (a), the final metrics (rounded to the nearest thousandth) are:

- Precision = 0.869.
- Recall = 0.864.
- F1 = 0.867.
- Exact Match = 0.441.
- Well Form = 1.000.
- Tree Match = 0.519.

Unsurprisingly, all predictions are well formed. This is due to the `Calculator` class computing this metric assuming the added tree is in fact well formed.

For part (b), the errors the classifier is doing in the results are minor and are most likely due to two main reasons: 1) sparsity of data, 2) the greedy decoding algorithm used. For example, consider the following pair of reference (R) and decoding (D) spans:

```
R : [SUB [SL:CATEGORY_EVENT the nutcracker]
D : [SUB the [SUB nutcracker ...]
```

With high probability, the play "The Nutcracker" is very sparsely present in the training corpus, if there is any example. Consequently, the classifier fails to label correctly that span. To observe the second source of errors, consider now the following spans:

```
R : [SUB services [SUB in philly ] ]
D : [SUB in [SL:LOCATION philly ] ]
```

Since the decoding CKY algorithm starts with the most likely label for "Philly", which is a location, then the mistake is carried over to the following level of decoding, which in turn labels the span "in philly" with `SUB`. Consequently, although the first case will most likely only be solved having more examples in the training corpus, the second one could be improved using beam search or a more sophisticated decoding algorithm.

For part (c), the idea of using `None` labels to unlabeled spans is to help the classifier detecting non branching spans (spans that do not correspond to a branch of the tree). To see this, note that without it, then the label dictionary will only include relevant labels, and the `UNK` and `Token` labels. Consequently, confronted with an non branching span, the classifier has three options:

1) Use a relevant label (and branch the tree unnecesarily).

2) Label the span as `UNK` (which also branches the tree).

3) Label as `Token` (which implies that the classifier expects a word).

Any classifier choosing 1) or 2) will make mistakes in the Tree Match and Exact Match metrics, which will render the model useless. Similarly, choosing 3) implies that during decoding, the tree will not continue in that branch, which also harms the previously mentioned metrics. On the contrary, if the tree is labeled as `None` for that span, then the decoding algorithm will skip this span, without branching or finishing the tree.