Introduction

Welcome to **Homework 3 - Trees**! In this homework, we will practice *parsing* on sentences from a semantic parsing corpus.

The data is obtained from this paper (see Figure 1). As you can see from the figure, the purpose of this task is to understand what are the users *intents* from a query in plain text.

The end goal is that given sentence to decode a binary **tree structure** with *semantic tags* as *nodes*. For example:

> whats there to do this weekend -> [IN:GET_EVENT whats there to do [SL:DATE_TIME this weekend]]

Note that the brackets [LABEL a substring of the text] indicates that this span is a sub-tree and LABEL is the semantic label of the root of the sub-tree. You might read more about bracket representation in this tutorial.

1. In **Part A**, we formulate this problem as a simple classification problem --- the input to the classifier will be `(text, span)` and the output will be the semantic `label` of that span. `span` is represented by two integer `(i,j)` which are the start and the end locations of the span.
2. In **Part B**, we will implement a **CKY**-style decoding algorithm to decode the final tree based on the classifier we trained in Part A.

We did pre-processing to enable CKY-style decoding for you. This includes binarization of the trees and handling of unary rules. (see the code).

Let's start by loading some dependencies and downloading the data as usual.

```
%%bash
!(stat -t /usr/local/lib/*/dist-packages/google/colab > /dev/null 2>&1) && exit
rm -rf hw3
git clone https://github.com/mit-6864/hw3.git
```

```
Cloning into 'hw3'...
```

```
import sys
sys.path.append("/content/hw3/trees/")
import numpy as np
import random
import torch
from torch import cuda
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
```

```
from span_tree import *
seed = 0
random.seed(seed)
np.random.seed(seed)
torch.manual_seed(seed)

if cuda.is_available():
  device = 'cuda'
  torch.cuda.manual_seed_all(seed)
else:
  print('WARNING: you are running this assignment on a cpu!')
  device = 'cpu'
```

Agenda

We apply a model that learns the parsing structures in 4 steps.

1. Enumerate all possible spans of a sentence
2. Generating word and span embeddings
3. Learning span label classifications
4. Decoding a tree structure using the classification distributions of spans

We go through this process step by step through the homework

PARTA

Data Processing

The very first step of the project is to load the corpus, building the **vocabulary**, **span label set**, and **span indices**.

We first need to enumerate every node of a tree with a Depth First Search (DFS).

```
def tree_dfs(node, span_list, label_dict, mode):
    """
    The base function for the recursion:
      node: current root while traversing the tree
      span_list: keep tracks of the spans an their label encodings in the tree e.g [[(0,1), 1
      label_dict: mapping from label to their encodings e.g {"UNK":0, "Token":1,"None":2, ...
      mode: "train" or "eval"
    """

    if len(node.children) == 0:
        assert(type(node) == Token)
        cur_span = (node.index, node.index + 1)
```

```
        cur_label = label_dict['Token']
        span_list.append([cur_span, cur_label])
        return span_list, label_dict

    cur_span = node.get_token_span()
    cur_label = node.label
    if node.label in label_dict:
        cur_label = label_dict[node.label]
    elif mode == 'train': # we are constructing the label dictionary
        cur_label = len(label_dict)
        label_dict[node.label] = cur_label
    else:
        cur_label = label_dict['UNK']
    span_list.append([cur_span, cur_label])

    if len(node.children) > 1: #if only has one child, we will ignore the Token label, otherw
        for child in node.children:
            # --------- Your code (hint: only need one single line) --------- #
            # span_list = tree_dfs(child, span_list, label_dict, mode)[0]
            tree_dfs(child, span_list, label_dict, mode)
            # --------- Your code ends --------- #
    return span_list, label_dict
```

Now, we go through the corpus and construct the **vocab dictionary** and the **label dictionary**. Note that we just adding new words and labels to the dictionaries while building the training set. Unseen words or labels in validation and test set are marked as unknown (UNK).

```
def process_line(line, vocab_dict, label_dict, mode):
    '''
    Processing a line in the corpus.
    line format: Sentence \t Sentence_Tree \n

    Example:
        'what is the shortest way home\t
        [IN:GET_DIRECTIONS what [SUB is [SUB the [SUB shortest [SUB way [SL:DESTINATION home

    Inputs:
    vocab_dict: vocab dictionary {word: word_index, ...}
    labels_dict: label dictionary {label: label_index, ...}
    '''
    s, s_tree = line.strip().split('\t')
    words = s.split(' ')
    word_ids = []
    for word in words:
        if word in vocab_dict:
            word_ids.append(vocab_dict[word])
        elif mode == 'train':
            word_ids.append(len(vocab_dict))
            vocab_dict[word] = len(vocab_dict)
        else:
```

```
        else:
            word_ids.append(vocab_dict['UNK'])

    tree = Tree(s_tree)
    span_list = []
    span_list, label_dict = tree_dfs(tree.root.children[0], span_list, label_dict, mode)
    return word_ids, span_list, vocab_dict, label_dict

def process_corpus(corpus_path, mode, vocab_dict=None, label_dict=None):
    lines = open(corpus_path).readlines()
    if not vocab_dict:
        vocab_dict = {'UNK': 0}
    if not label_dict:
        label_dict = {'UNK': 0, 'Token': 1, 'None': 2}
    corpus = []
    sent_spans = []
    raw_lines = []
    for line in lines:
      if len(line.strip()) < 3:
        continue
      word_ids, span_list, vocab_dict, label_dict = process_line(line, vocab_dict, label_dict
      corpus.append(word_ids)
      sent_spans.append(span_list)
      raw_lines.append(line)
    return corpus, sent_spans, vocab_dict, label_dict, raw_lines
```

```
corpus_train, spans_train, vocab_dict, label_dict, train_lines = process_corpus('/content/hw3
corpus_valid, spans_valid, _, _, valid_lines = process_corpus('/content/hw3/trees/valid.txt',
                                                 vocab_dict=vocab_dict, label_dict=label_dict
corpus_test,  spans_test, _, _, test_lines = process_corpus('/content/hw3/trees/test.txt', 'e
                                                 vocab_dict=vocab_dict, label_dict=label_dict
```

```
inv_vocab_dict = np.array(list(vocab_dict.keys()))
inv_label_dict = np.array(list(label_dict.keys()))
```

```
num_words = len(vocab_dict)
num_labels = len(label_dict)

print('Number of different words: {}'.format(num_words))
print('Number of different labels: {}'.format(num_labels))
```

```
Number of different words: 8626
Number of different labels: 147
```

Let see how the data looks like, and compare with our output in below:

```
['how', 'long', 'will', 'it', 'take', 'to', 'drive', 'from', 'chicago', 'to', 'mississippi']
how long will it take to drive from chicago to mississippi    [IN:GET_ESTIMATED_DURATION how [SUB lo

[[(0, 11), 3], [(0, 1), 1], [(1, 11), 4], [(1, 2), 1], [(2, 11), 4], [(2, 3), 1], [(3, 11), 4], [(3,
['will', 'it', 'take', 'shorter', 'to', 'get', 'to', 'the', 'white', 'house', 'by', 'bus', 'or', 'ta
will it take shorter to get to the white house by bus or taxi ?    [IN:UNSUPPORTED_NAVIGATION will [

[[(0, 15), 8], [(0, 1), 1], [(1, 15), 4], [(1, 2), 1], [(2, 15), 4], [(2, 3), 1], [(3, 15), 4], [(3,
['will', 'i', 'make', 'it', 'to', 'the', 'beach', 'by', 'noon', 'if', 'i', 'leave', 'now']
will i make it to the beach by noon if i leave now    [IN:GET_ESTIMATED_ARRIVAL will [SUB i [SUB mak

[[(0, 13), 9], [(0, 1), 1], [(1, 13), 4], [(1, 2), 1], [(2, 13), 4], [(2, 3), 1], [(3, 13), 4], [(3,
['when', 'should', 'i', 'leave', 'my', 'house', 'to', 'get', 'to', 'the', 'hamilton', 'mall', 'right
when should i leave my house to get to the hamilton mall right when it opens on saturday    [IN:GET_

[[(0, 18), 13], [(0, 1), 1], [(1, 18), 4], [(1, 2), 1], [(2, 18), 4], [(2, 3), 1], [(3, 18), 4], [(3
['i', 'need', 'to', 'know', 'if', 'there', "'s", 'a', 'lot', 'of', 'traffic', 'on', 'my', 'way', 'ho
i need to know if there 's a lot of traffic on my way home    [IN:GET_INFO_TRAFFIC i [SUB need [SUB

[[(0, 15), 17], [(0, 1), 1], [(1, 15), 4], [(1, 2), 1], [(2, 15), 4], [(2, 3), 1], [(3, 15), 4], [(3
```

```
for i in range(5):
  print([inv_vocab_dict[w] for w in corpus_train[i]])
  print(train_lines[i])
  print(spans_train[i])
```

```
['how', 'long', 'will', 'it', 'take', 'to', 'drive', 'from', 'chicago', 'to', 'mississi
how long will it take to drive from chicago to mississippi      [IN:GET_ESTIMATED_DURAT

[[(0, 11), 3], [(0, 1), 1], [(1, 11), 4], [(1, 2), 1], [(2, 11), 4], [(2, 3), 1], [(3,
['will', 'it', 'take', 'shorter', 'to', 'get', 'to', 'the', 'white', 'house', 'by', 'bu
will it take shorter to get to the white house by bus or taxi ? [IN:UNSUPPORTED_NAVIGAT

[[(0, 15), 8], [(0, 1), 1], [(1, 15), 4], [(1, 2), 1], [(2, 15), 4], [(2, 3), 1], [(3,
['will', 'i', 'make', 'it', 'to', 'the', 'beach', 'by', 'noon', 'if', 'i', 'leave', 'no
will i make it to the beach by noon if i leave now      [IN:GET_ESTIMATED_ARRIVAL will

[[(0, 13), 9], [(0, 1), 1], [(1, 13), 4], [(1, 2), 1], [(2, 13), 4], [(2, 3), 1], [(3,
['when', 'should', 'i', 'leave', 'my', 'house', 'to', 'get', 'to', 'the', 'hamilton', '
when should i leave my house to get to the hamilton mall right when it opens on saturda

[[(0, 18), 13], [(0, 1), 1], [(1, 18), 4], [(1, 2), 1], [(2, 18), 4], [(2, 3), 1], [(3,
['i', 'need', 'to', 'know', 'if', 'there', "'s", 'a', 'lot', 'of', 'traffic', 'on', 'my
i need to know if there 's a lot of traffic on my way home      [IN:GET_INFO_TRAFFIC i

[[(0, 15), 17], [(0, 1), 1], [(1, 15), 4], [(1, 2), 1], [(2, 15), 4], [(2, 3), 1], [(3,
```

Defining the Neural Network

Sentence Encoding

We use a Bi-directional LSTM for sentence encoding. We build a sentence encoder with a embedding layer and a Bi-directional LSTM layer:

- Input:
  - word indices: `[batch_size, sentence_length]`
- Output:
  - word embeddings: `[batch_size, sentence_length, hidden_size]`

```
class SentEnc(nn.Module):
    def __init__(self, num_words, num_layers, hidden_size, dropout=0):
        super(SentEnc, self).__init__()
        self.num_words = num_words
        self.num_layers = num_layers
        self.hidden_size = hidden_size
        self.dropout = dropout
        self.embedding = nn.Embedding(num_words, hidden_size)
        # --------- Your code --------- #
        # Construct your lstm module here (single line):
        self.lstm = nn.LSTM(input_size=hidden_size, hidden_size=hidden_size, dropout=dropout,


        # --------- Your code ends --------- #

    def forward(self, x):
        # --------- Your code --------- #
        embedding = self.embedding(x)
        bidirectional, _ = self.lstm(embedding) # assuming batch size = 1
        # --------- Your code ends --------- #
        return bidirectional
```

Span Encodings

Given the LSTM outputs, we generate the span embeddings with the span indices.

We generate a span embedding by concatenating the word embeddings of the first and last words of a span. For example, if a span starts from the i-th word and ends at the j-th word, our span embedding would be

$$[h_i^T; h_j^T]^T$$

where h_i stands for the Bi-LSTM output of the i^{th} word.

In Pytorch, Given the hidden states $h[0], h[1], \ldots, h[n]$, where

```
h[i].size() = [1, k]
```

the embedding of span (i, j) is

```
span_ij = torch.cat([h[i], h[j]], dim=1)
span_ij.size() = [1, 2 * k]
```

Please complete the following function for generating span embeddings.

- Input:
  - word embeddings: `[sentence_length, hidden_size]`
  - span indices: `[num_span, 2]`
- Output:
  - span embeddings `[num_span, hidden_size * 2]`

```
def get_span_embeddings(word_embeddings, span_indices):
    word_embeddings = word_embeddings[0]
    span_embeddings = torch.empty(span_indices.size(0), word_embeddings.size(1)*2).to(device)
    # print('Word embeddings', word_embeddings.shape)
    for idx, (i,j) in enumerate(span_indices):
        i, j = i.item(), j.item()
        span_embeddings[idx] = torch.cat([word_embeddings[i], word_embeddings[j-1]])
    return span_embeddings
```

Tag Prediction

We build a Classifier that puts the neural models together. The classifier takes word and span indices as inputs, and predict span labels by calculating word embeddings, span embeddings, and label logits. we will predict the tag of the spans with a linear classifier.

- Inputs:
  - word indices: `[batch_size, num_words]`
- Outputs:
  - span predictions: `[num_spans, num_labels]`

Please implement the forward function following 4 steps:

1. Generate the word embeddings by processing the input sentences with the LSTM sentence encoder.
2. Apply dropout on word embeddings.
3. Calculate span embeddings with function get_span_embeddings().
4. Calculate label logits with the linear layer defined as follows.

```
class Classifier(nn.Module):

    def __init__(self, num_words, num_labels, num_layers, hidden_size, dropout=0):
        super(Classifier, self).__init__()
        self.sent_enc = SentEnc(num_words, num_layers, hidden_size)
        self.dropout = nn.Dropout(dropout)
        self.linear = nn.Linear(4 * hidden_size, num_labels)
        # self.softmax = nn.Softmax(dim=-1)

    def forward(self, x, span_indices):
        # --------- Your code --------- #
        embeddings = self.sent_enc(x)
        embeddings = self.dropout(embeddings)
        span_embeddings = get_span_embeddings(embeddings, span_indices)
        logits = self.linear(span_embeddings.float().to(device))
        # --------- Your code ends --------- #
        return logits


#For decoding, we add some random spans and label them as "None"
def add_none_span(word_list, span_list, label_dict, all=False):
    num_words = len(word_list)
    num_labeled_span = len(span_list)
    labeled_span_set = set([span for span, label in span_list])
    none_spans = []
    for i in range(num_words):
        for j in range(i + 1, num_words):
            if (i, j) not in labeled_span_set:
                none_spans.append([(i, j), label_dict['None']])
    if not all:
        k = min(num_labeled_span, len(none_spans))
        sampled_none_spans = random.sample(none_spans, k)
    else:
        sampled_none_spans = none_spans
    return span_list + sampled_none_spans
```

Training Loop

With all neural models already defined, we are implementing the training loop.

```
print('Using device: {}'.format(device))
```

```
# just remeber you can tune these hyper-parameters!
batch_size = 1
num_layers = 2
hidden_size = 200
lr = 0.05
num_epochs = 3 #Be aware of over-fitting!
loss_fn = nn.CrossEntropyLoss().to(device)
dropout = 0.25

classifier = Classifier(num_words, num_labels, num_layers, hidden_size, dropout)
optimizer = optim.SGD(classifier.parameters(), lr=lr, momentum=0.9)

classifier = classifier.to(device)
classifier.train()

for epoch in range(num_epochs):
    total_loss = 0
    classifier.train()
    for i in range(len(corpus_train)):

        if i % 10000 == 0:
            print('Epoch {} Batch {}'.format(epoch, i))

        cur_spans = add_none_span(corpus_train[i], spans_train[i], label_dict)

        sent_inputs  = torch.Tensor([corpus_train[i]]).long().to(device)
        span_indices = torch.Tensor([x[0] for x in cur_spans]).long().to(device)
        span_labels  = torch.Tensor([x[1] for x in cur_spans]).long().to(device)

        # --------- Your code --------- #
        classifier.zero_grad()
        logits = classifier(sent_inputs, span_indices)
        loss = loss_fn(logits, span_labels)
        total_loss += loss.data.item()
        loss.backward()
        optimizer.step()
        # --------- Your code ends --------- #

    print('Epoch {}, train loss={}'.format(epoch, total_loss / len(corpus_train)))

    total_loss = 0
    classifier.eval()
    for i in range(len(corpus_valid)):
        #if i % 10000 == 0:
        #    print('Epoch {} Batch {}'.format(epoch, i))
        cur_spans = add_none_span(corpus_valid[i], spans_valid[i], label_dict)
        sent_inputs  = torch.Tensor([corpus_valid[i]]).long().to(device)
        span_indices = torch.Tensor([x[0] for x in cur_spans]).long().to(device)
        span_labels  = torch.Tensor([x[1] for x in cur_spans]).long().to(device)
        # --------- Your code --------- #
```

```
        with torch.no_grad():
            logits = classifier(sent_inputs, span_indices)
            loss = loss_fn(logits, span_labels)
            total_loss += loss.data.item()
        # --------- Your code ends --------- #
    print('Epoch {}, valid loss={}'.format(epoch, total_loss / len(corpus_valid)))
```

```
Using device: cuda
Epoch 0 Batch 0
Epoch 0 Batch 10000
Epoch 0 Batch 20000
Epoch 0 Batch 30000
Epoch 0, train loss=0.19727645829572407
Epoch 0, valid loss=0.10192600763689287
Epoch 1 Batch 0
Epoch 1 Batch 10000
Epoch 1 Batch 20000
Epoch 1 Batch 30000
Epoch 1, train loss=0.09120403574272427
Epoch 1, valid loss=0.08135060106000994
Epoch 2 Batch 0
Epoch 2 Batch 10000
Epoch 2 Batch 20000
Epoch 2 Batch 30000
Epoch 2, train loss=0.06110643244547971
Epoch 2, valid loss=0.07863004592277803
```

Evaluation

After training the model, we evaluate the classification results.
What we will do is that we treat a tree strcture as a bag of spans (a list of span indices), and then compute F-1 score.

```
from itertools import zip_longest
from typing import Counter, Dict, Optional
import numpy as np

class Calculator:
    def __init__(self, strict = False) -> None:
        self.TP = 0
        self.gold_P = 0
        self.pred_P = 0
        self.exact_match = []
        self.tree_match = []
        self.well_form = []
        self.strict = strict

    def get_metrics(self):
        precision = (self.TP / self.pred_P) if self.pred_P else 0
        recall = (self.TP / self.gold_P) if self.gold_P else 0
        f1 = (2.0 * precision * recall / (precision + recall)) if (precision + recall) else 0
```

```
        return {
            "precision": precision,
            "recall": recall,
            "f1": f1,
            "exact_match": np.mean(self.exact_match),
            "well_form": np.mean(self.well_form),
            "tree_match":  np.mean(self.tree_match),
            "num_examples": len(self.exact_match)
        }

    def is_well_formed(self, spans):
        for s1 in spans:
          for s2 in spans:
              if s1[0] < s2[0] and s2[0] < s1[1] and s1[1] < s2[1]:
                    return False
        return True

    def add_instance_span(self, gold_spans, pred_spans):
        self.gold_P += len(gold_spans)
        self.pred_P += len(pred_spans)
        self.TP += len(set(gold_spans) & set(pred_spans))
        self.exact_match.append(int(set(gold_spans) == set(pred_spans)))
        gold_spans = [s[0] for s in gold_spans]
        pred_spans = [s[0] for s in pred_spans]
        self.tree_match.append(int(set(gold_spans) == set(pred_spans)))
        well_formed = self.is_well_formed(pred_spans)
        self.well_form.append(int(well_formed))

    def add_instance_tree(self, gold_tree, pred_tree):
        node_info_gold = self._get_node_info(gold_tree)
        self.gold_P += len(node_info_gold)
        node_info_pred = self._get_node_info(pred_tree)
        self.pred_P += len(node_info_pred)
        self.TP += len(node_info_gold & node_info_pred)
        self.exact_match.append(int(node_info_gold.keys() == node_info_pred.keys()))
        self.well_form.append(1) #we assume the decoded tree is indeed a tree :)
        node_info_gold = {k[1] for k,v in node_info_gold.items()}
        node_info_pred = {k[1] for k,v in node_info_pred.items()}
        self.tree_match.append(int(node_info_gold==node_info_pred))

    def _get_node_info(self, tree) -> Counter:
        nodes = tree.root.list_nonterminals()
        node_info: Counter = Counter()
        for node in nodes:
            if node.label != 'Token':
              span = self._get_span(node)
              node_info[(node.label, self._get_span(node))] += 1

        return node_info

    def _get_span(self, node):
```

```
    def _get_span(self, node):
        return node.get_flat_str_spans(
        ) if self.strict else node.get_token_span()


classifier.eval()
parta_calc = Calculator(strict=False)
pred_bag_spans = []
gold_bag_spans = []
for (tokens, spans, line) in zip(corpus_test,spans_test,test_lines):
    #We only test non-Token labels
    spans = [tuple(x) for x in spans if x[1] != 1]

    if len(spans) <= 1 or len(line.strip()) < 3:
      continue

    all_spans = [(i,j) for i in range(len(tokens))
                      for j in range(i + 1, len(tokens) + 1)]

    input  = torch.Tensor([tokens]).long().to(device)
    logits = classifier(input, torch.Tensor(all_spans).long().to(device))

    pred_spans = []
    for i, span in enumerate(all_spans):
        label_idx = torch.argmax(logits[i]).item()
        if label_idx != 2 and label_idx != 1:
          pred_spans.append((span,label_idx))

    parta_calc.add_instance_span(spans, pred_spans)
    pred_bag_spans.append(pred_spans)
    gold_bag_spans.append(spans)

print(parta_calc.get_metrics())
```

```
{'precision': 0.8464562801425876, 'recall': 0.9467160731246849, 'f1': 0.893783315528149
```

PARTB (Only for 6.864 students)

The remaining will be **PartB** for **HW3-Trees**.
In PartB, we will decode a tree based on the classifier trained on part A.

CKY

You will be implementing the following simple CYK recursion:

`best_score[i,j]=max_k {best_score[i,k]+best_score[k,j]} + max_l {span_dict[(i,j)][l]}`

where `l` is the label of the current span `(i,j)`, and `k` is the splitting point

Note that this is a simpler recursion than the full CKY algorithm.

```
from torch.nn.functional import log_softmax
EPS = 1e-6
dp_results = []
classifier.eval()
for kk,(line,spans,tokens) in enumerate(zip(test_lines,spans_test,corpus_test)):
    spans = [tuple(x) for x in spans if  x[1] != 1]

    if len(spans) <= 1 or len(line.strip()) < 3:
      continue

    sent_inputs  = torch.Tensor([tokens]).long().to(device)

    all_spans = [(i,j) for i in range(len(tokens))
                     for j in range(i + 1, len(tokens) + 1)]

    logits = classifier(sent_inputs, torch.Tensor(all_spans).long().to(device))
    logprobs = log_softmax(logits, dim = -1)
    # span dict will map each span (l,r) to its predicted distribution of labels
    span_dict = {}
    for i, s in enumerate(all_spans):
      span_dict[s]  = logprobs[i]

    TOKEN_ID, NULL_ID = 1, 2
    best_score, best_split, best_label = {}, {}, {} # we will do dynamic programming to decod
    # Think: why do we first iterate the length of the span?
    for ll in range(1, len(tokens) + 1): # length of the span
        for i in range(0, len(tokens)-ll+1): # start of the span
            j = i + ll
            cur_span = (i, j)
            if j == i + 1:
                span_dict[cur_span][NULL_ID]  = -1/EPS
                # --------- Your code --------- #
                #use span_dict[cur_span] to update best_label and best_score, be careful, it
                best_score[cur_span] = span_dict[cur_span].max().item()
                best_label[cur_span] = torch.argmax(span_dict[cur_span]).item()

                # --------- Your code ends --------- #
                best_split[cur_span] = None
            else:
                span_dict[cur_span][NULL_ID]  = -1/EPS # we will never decode a NULL sub-tree
                span_dict[cur_span][TOKEN_ID] = -1/EPS # we will never decode a NULL sub-tree
                # --------- Your code --------- #
                #try to give the values for best_score/label/split[cur_span]
                best_score[cur_span] = float('-inf')
                """"best_score[i,j]=max_k {best_score[i,k]+best_score[k,j]} + max_l {span_dic
                for k in range(i+1, j):
                    score = best_score[(i,k)] + best_score[(k,j)]
                    if score > best_score[cur_span]:
                        best_score[cur_span] = score
```

```
                    best_score[cur_span] = score
                    best_split[cur_span] = k
                best_score[cur_span] += torch.max(span_dict[cur_span]).item()
                best_label[cur_span] = torch.argmax(span_dict[cur_span]).item()


                # --------- Your code ends --------- #
    dp_results.append((best_score, best_split, best_label))
print(len(dp_results))
```

```
8997
```

Tree Construction

In this section, we will construct a tree using the DP results.
Before start doing it, please get yourself a little familiar with the span_tree.py.

```
import sys
def get_nodetype(label):
    if label.startswith(PREFIX_INTENT):
        node = Intent(label)
    elif label.startswith(PREFIX_SLOT):
        node = Slot(label)
    elif label.startswith(PREFIX_SUBTREE):
        node = SubTree(label)
    else:
        print('something wrong with the label!!!', label)
        sys.error()
    return node

def dfs_build(l, r, best_label, best_split):
    if l + 1 == r:
        la = best_label[(l,r)]
        if la == 1:
            return Token(surface_tokens[l], l)
        else:
            node = get_nodetype(inv_label_dict[la])
            node.children = [Token(surface_tokens[l], l)]
            node.children[0].parent = node
            return node

    label = inv_label_dict[best_label[(l, r)]]
    node = get_nodetype(label)

    #--- your code --- #
    #hint: use best_split! and recursion to assign node.children here
    k = best_split[(l,r)]
    node.children = [dfs_build(l, k, best_label, best_split), dfs_build(k, r, best_label, bes
    #--- your code ends --- #
```

```
        for c in node.children:
            c.parent = node

        return node



    pred_trees = []
    gold_trees = []
    partb_calc = Calculator(strict=False)
    k = 0
    for i,(line,spans,tokens) in enumerate(zip(test_lines,spans_test,corpus_test)):
        surface_tokens, str_ref_tree = line.strip().split('\t')
        surface_tokens = surface_tokens.split()
        spans = [tuple(x) for x in spans if x[1] != 1]

        if len(spans) <= 1 or len(line.strip()) < 3:
          continue

        best_score, best_split, best_label = dp_results[k]
        k+=1
        root = Root()
        root.children = [dfs_build(0, len(tokens), best_label, best_split)]
        root.children[0].parent = root
        tree = Tree('IN:GET_EVENT placeholder') #the string here is just a placeholder
        tree.root = root
        if k < 10: #use this info for debugging! Does your tree make sense?
            print(k, line.strip())
            print('REF:', str_ref_tree)
            print('DEC:', str(tree))
            print()
        """ here's some decoding examples we get
          1 whats there to do this weekend  [IN:GET_EVENT whats [SUB there [SUB to [SUB do [SL:DA
          REF: [IN:GET_EVENT whats [SUB there [SUB to [SUB do [SL:DATE_TIME this weekend ] ] ] ]
          DEC: [IN:GET_EVENT whats [SUB there [SUB to [SUB do [SL:DATE_TIME this weekend ] ] ] ]

          2 what is a good restaurant for tex mex in austin [IN:UNSUPPORTED what [SUB is [SUB a [
          REF: [IN:UNSUPPORTED what [SUB is [SUB a [SUB good [SUB restaurant [SUB for [SUB tex [S
          DEC: [IN:UNSUPPORTED what [SUB is [SUB a [SUB good [SUB restaurant [SUB for [SUB tex [S

          3 where can i see the fireworks tonight [IN:GET_EVENT where [SUB can [SUB i [SUB see [S
          REF: [IN:GET_EVENT where [SUB can [SUB i [SUB see [SUB [SL:CATEGORY_EVENT the fireworks
          DEC: [IN:GET_EVENT where [SUB can [SUB i [SUB see [SUB the [SUB fireworks [SL:DATE_TIME
        """
        partb_calc.add_instance_tree(Tree(str_ref_tree), tree)
        pred_trees.append(tree)
        gold_trees.append(Tree(str_ref_tree))
```

```
 1 whats there to do this weekend        [IN:GET_EVENT whats [SUB there [SUB to [SUB do
 REF: [IN:GET_EVENT whats [SUB there [SUB to [SUB do [SL:DATE_TIME this weekend ] ] ] ]
 DEC: [IN:GET_EVENT whats [SUB there [SUB to [SUB do [SL:DATE_TIME this weekend ] ] ] ]
```

```
2 what is a good restaurant for tex mex in austin       [IN:UNSUPPORTED what [SUB is [S
REF: [IN:UNSUPPORTED what [SUB is [SUB a [SUB good [SUB restaurant [SUB for [SUB tex [S
DEC: [IN:UNSUPPORTED_EVENT what [SUB is [SUB a [SUB good [SUB restaurant [SUB for [SUB

3 where can i see the fireworks tonight [IN:GET_EVENT where [SUB can [SUB i [SUB see [S
REF: [IN:GET_EVENT where [SUB can [SUB i [SUB see [SUB [SL:CATEGORY_EVENT the fireworks
DEC: [IN:GET_EVENT where [SUB can [SUB i [SUB see [SUB the [SUB fireworks [SL:DATE_TIME

4 restaurants offering prefixed menus in midtown        [IN:UNSUPPORTED restaurants [SU
REF: [IN:UNSUPPORTED restaurants [SUB offering [SUB prefixed [SUB menus [SUB in midtown
DEC: [IN:UNSUPPORTED restaurants [SUB offering [SUB prefixed [SUB menus [SUB in midtown

5 where should i go dancing this weekend        [IN:GET_EVENT where [SUB should [SUB i
REF: [IN:GET_EVENT where [SUB should [SUB i [SUB go [SUB [SL:CATEGORY_EVENT dancing ] [
DEC: [IN:GET_EVENT where [SUB should [SUB i [SUB go [SUB [SL:CATEGORY_EVENT dancing ] [

6 what is going on this weekend [IN:GET_EVENT what [SUB is [SUB going [SUB on [SL:DATE_
REF: [IN:GET_EVENT what [SUB is [SUB going [SUB on [SL:DATE_TIME this weekend ] ] ] ] ]
DEC: [IN:GET_EVENT what [SUB is [SUB going [SUB on [SL:DATE_TIME this weekend ] ] ] ] ]

7 are there any santa meetings this weekend     [IN:GET_EVENT are [SUB there [SUB any [
REF: [IN:GET_EVENT are [SUB there [SUB any [SUB [SL:CATEGORY_EVENT santa meetings ] [SL
DEC: [IN:GET_EVENT are [SUB there [SUB any [SUB santa [SUB [SL:CATEGORY_EVENT meetings

8 what breakfast locations within 5 miles of me open at 6 am    [IN:UNSUPPORTED what [S
REF: [IN:UNSUPPORTED what [SUB breakfast [SUB locations [SUB within [SUB 5 [SUB miles [
DEC: [IN:GET_EVENT what [SUB breakfast [SUB locations [SUB within [SUB 5 [SUB miles [SU

9 what can i do with my friends tomorrow night  [IN:GET_EVENT what [SUB can [SUB i [SUB
REF: [IN:GET_EVENT what [SUB can [SUB i [SUB do [SUB with [SUB [SL:ATTENDEE_EVENT--IN:G
DEC: [IN:GET_EVENT what [SUB can [SUB i [SUB do [SUB with [SUB my [SUB friends [SL:DATE
```

```
print(partb_calc.get_metrics())
```

```
{'precision': 0.8707169525970037, 'recall': 0.8641517841438103, 'f1': 0.867421946267251
```

Recommended Reading (not required, just for interested students):

https://arxiv.org/pdf/1810.07942.pdf

https://www.aclweb.org/anthology/D16-1257/

https://arxiv.org/abs/1412.7449