Task B: Named Entity Recognition with CRF on Hindi Dataset. (Total: 60 Points out of 100)

In this part, you will use a CRF to implement a named entity recognition tagger. We have implemented a CRF for you in crf.py along with some functions to build, and pad feature vectors. Your job is to add more features to learn a better tagger. Then you need to complete the traiing loop implementation.

Finally, you can checkout the code in `crf.py` -- reflect on CRFs and span tagging, and answer the discussion questions.

We will use the Hindi NER dataset at: https://github.com/cfiltnlp/HiNER

The first step would be to download the repo into your current folder of the Notebook

In [1]:
```
#!git clone https://github.com/cfiltnlp/HiNER.git
```

In [2]:
```
import torch
```

In [3]:
```
# This is so that you don't have to restart the kernel everytime you edit hmm.py

%load_ext autoreload
%autoreload 2
```

First we load the data and labels. Feel free to explore them below.

Since we have provided a seperate train and dev split, there is not need to split the data yourself.

In [4]:
```
from crf import load_data, make_labels2i

train_filepath = "./HiNER/data/collapsed/train.conll"
dev_filepath = "./HiNER/data/collapsed/validation.conll"
labels_filepath = "./HiNER/data/collapsed/label_list"

train_sents, train_tag_sents = load_data(train_filepath)
dev_sents, dev_tag_sents = load_data(dev_filepath)
labels2i = make_labels2i(labels_filepath)

print("train sample", train_sents[2], train_tag_sents[2])
print()
print("labels2i", labels2i)
```

```
train sample ['रामनगर', 'इगलास', ',', 'अलीगढ़', ',', 'उत्तर', 'प्रदेश', 'स्थित', 'एक', 'गाँव', 'है।'] ['B-LOCATION', 'B-LOCA
TION', 'O', 'B-LOCATION', 'O', 'B-LOCATION', 'I-LOCATION', 'O', 'O', 'O', 'O']

labels2i {'<PAD>': 0, 'B-LOCATION': 1, 'B-ORGANIZATION': 2, 'B-PERSON': 3, 'I-LOCATION': 4, 'I-ORGANIZATION':
5, 'I-PERSON': 6, 'O': 7}
```

Feature engineering. (Total 30 points)

Notice that we are **learning** features to some extent: we start with one unique feature for every possible word. You can refer to figure 8.15 in the textbook for some good baseline features to try.

identity of w_i, identity of neighboring words
embeddings for w_i, embeddings for neighboring words
part of speech of w_i, part of speech of neighboring words
presence of w_i in a **gazetteer**
w_i contains a particular prefix (from all prefixes of length ≤ 4)
w_i contains a particular suffix (from all suffixes of length ≤ 4)
word shape of w_i, word shape of neighboring words
short word shape of w_i, short word shape of neighboring words
gazetteer features

Figure 8.15 Typical features for a feature-based NER system.

There is no need to worry about embeddings now.

Hindi POS Tagger (10 Points)

Although this step is not entirely necessary, if you want to use the HMM pos tagger to extract feature corresponding to the pos of the word in the sentence, we need to add this into the pipeline.

You get 10 points if you use your pos_tagger to featurize the sentences

In [5]:
```
from hmm import get_hindi_dataset
import pickle
from typing import List

words, tags, observation_dict, state_dict, all_observation_ids, all_state_ids = get_hindi_dataset()

# we need to add the id for unknown word (<unk>) in our observations vocab
UNK_TOKEN = '<unk>'

observation_dict[UNK_TOKEN] = len(observation_dict)
print("id of the <unk> token:", observation_dict[UNK_TOKEN])

## load the pos tagger
pos_tagger = pickle.load(open('hindi_pos_tagger.pkl', 'rb'))

def encode(sentences: List[List[str]]) -> List[List[int]]:
    """
    Using the observation_dict, convert the tokens to ids
    unknown words take the id for UNK_TOKEN
    """
    return [
        [observation_dict[t] if t in observation_dict else observation_dict[UNK_TOKEN]
            for t in sentence]
        for sentence in sentences]

def get_pos(pos_tagger, sentences) -> List[List[str]]:
    """
    The the pos tag for input sentences
    """
    sentence_ids = encode(sentences)
    decoded_pos_ids = pos_tagger.decode(sentence_ids)
    return [
        [tags[int(i)] for i in d_ids]
        for d_ids in decoded_pos_ids
    ]
```

```
[nltk_data] Downloading package indian to
[nltk_data]     /Users/xingyuchen/nltk_data...
[nltk_data]   Package indian is already up-to-date!
id of the <unk> token: 2186
```

Feature Engineering Functions (20 Points)

In [48]:
```
from typing import List
import numpy as np
# TODO: Update this function to add more features
#      You can check crf.py for how they are encoded, if interested.

with open('gazetteer_hindi.txt', 'r', encoding= 'utf-16') as f:
    gazetteer = f.read()
with open('hindi_suffixes.txt', 'r', encoding= 'utf-8') as f:
    suffixes = f.read()

def make_features(text: List[str]) -> List[List[int]]:
    """Turn a text into a feature vector.

    Args:
        text (List[str]): List of tokens.

    Returns:
        List[List[int]]: List of feature Lists.
    """
    feature_lists = []
    for i, token in enumerate(text):
        feats = []
        # We add a feature for each unigram.
        feats.append(f"word={token}")
        # TODO: Add more features here
        feats.append(f"pos={get_pos(pos_tagger, token)[0]}")
        if len(text) == 1:
            feats.append(f"prev_word={'<S>'}")
            feats.append(f"next_word={'<E>'}")
        else:
            if i == 0:
                feats.append(f"prev_word={'<S>'}")
                feats.append(f"next_word={text[i+1]}")
#                 feats.append(f"prev_pos={'<S>'}")
#                 feats.append(f"next_pos={get_pos(pos_tagger, text[i+1])[0]}")
                prev_word = token
            elif i == len(text) - 1:
                feats.append(f"prev_word={prev_word}")
#                 feats.append(f"prev_pos={get_pos(pos_tagger, text[i-1])[0]}")
                feats.append(f"next_word={'<E>'}")
#                 feats.append(f"next_pos={'<E>'}")
            else:
                feats.append(f"prev_word={prev_word}")
#                 feats.append(f"prev_pos={get_pos(pos_tagger, text[i-1])[0]}")
                feats.append(f"next_word={text[i+1]}")
#                 feats.append(f"next_pos={get_pos(pos_tagger, text[i+1])[0]}")
                prev_word = token
        if token in gazetteer:
            feats.append(f"gazetteer={'1'}")
        else:
            feats.append(f"gazetteer={'0'}")
        if list(filter(token.endswith, suffixes)) != []:
            feats.append(f"suffixes={set(list(filter(token.endswith, suffixes)))}")
        else:
            feats.append(f"suffixes={set(['NaN'])}")
        # We append each feature to a List for the token.
#         special_characters = "!\"#$%&'()*+,-./:;<=>?@[\]^_`{|}~"
#         if any(c in special_characters for c in token):
#             feats.append(f"specialsymbol={token}")
#         else:
#             feats.append(f"specialsymbol={'NaN'}")
#         print(feats)
        feature_lists.append(feats)
    return feature_lists
```

In [20]:
```
def featurize(sents: List[List[str]]) -> List[List[List[str]]]:
    """Turn the sentences into feature Lists.

    Eg.: For an input of 1 sentence:
         [[['I','am','a','student','at','CU','Boulder']]]
        Return list of features for every token for every sentence like:
        [[
         ['word=I',  'prev_word=<S>','pos=PRON',...],
         ['word=an', 'prev_word=I'  , 'pos=VB' ,...],
         [...]
        ]]

    Args:
        sents (List[List[str]]): A List of sentences, which are Lists of tokens.

    Returns:
        List[List[List[str]]]: A List of sentences, which are Lists of feature Lists
    """
    feats = []
    for sent in tqdm(sents):
        # Gets a List of Lists of feature strings
        feats.append(make_features(sent))
    return feats
```

Finish the training loop. (10 Points)

See the previous homework, and fill in the missing parts of the training loop.

In [51]:
```
from crf import f1_score, predict, PAD_SYMBOL, pad_features, pad_labels
import numpy as np
from tqdm.autonotebook import tqdm
import random
# TODO: Implement the training loop
# HINT: Build upon what we gave you for HW2.
# See cell below for how we call this training loop.

def training_loop(
    num_epochs,
    batch_size,
    train_features,
    train_labels,
    dev_features,
    dev_labels,
    optimizer,
    model,
    labels2i,
    pad_feature_idx
):
    samples = list(zip(train_features, train_labels))
    random.shuffle(samples)
    batches = []
    for i in range(0, len(samples), batch_size):
        batches.append(samples[i:i+batch_size])
    print("Training...")
    for i in range(num_epochs):
        losses = []
        for batch in tqdm(batches):
            # Here we get the features and labels, pad them,
            # and build a mask so that our model ignores PADs
            # We have abstracted the padding from you for simplicity,
            # but please reach out if you'd like learn more.
            features, labels = zip(*batch)
            features = pad_features(features, pad_feature_idx)
            features = torch.stack(features)
            # Pad the label sequences to all be the same size, so we
            # can form a proper matrix.
            labels = pad_labels(labels, labels2i[PAD_SYMBOL])
            labels = torch.stack(labels)
            mask = (labels != labels2i[PAD_SYMBOL])
            # TODO: Empty the dynamic computation graph
            optimizer.zero_grad()
            # TODO: Run the model. Since we use the pytorch-crf model,
            # our forward function returns the positive log-likelihood already.
            # We want the negative log-likelihood. See crf.py forward method in NERTagger
            loss = model.forward(features, labels, mask)
            # TODO: Backpropogate the loss through our model
            loss.backward()
            # TODO: Update our coefficients in the direction of the gradient.
            optimizer.step()
            # TODO: Store the losses for logging
            losses.append(loss.item())
        # TODO: Log the average Loss for the epoch
        print(f"epoch {i}, loss: {np.log(sum(losses)/len(losses))}")
        # TODO: make dev predictions with the `predict()` function
        dev_pred = predict(model, dev_features)
#         print('pred',dev_pred)
#         print('label', dev_labels)
        # TODO: Compute F1 score on the dev set and log it.
        print(f"Dev F1 log {np.log(f1_score(dev_pred, dev_labels, labels2i['O']))}")

    # Return the trained model
    return model
```

Run the training loop (10 Points)

We have provided the code here, but you can try different hyperparameters and test multiple runs.

In [49]:
```
from crf import build_features_set
from crf import make_features_dict
from crf import encode_features, encode_labels
from crf import NERTagger

# Build the model and featurized data
train_features = featurize(train_sents)
dev_features = featurize(dev_sents)

# Get the full inventory of possible features
all_features = build_features_set(train_features)
# Hash all features to a unique int.
features_dict = make_features_dict(all_features)
# Initialize the model.
model = NERTagger(len(features_dict), len(labels2i))

encoded_train_features = encode_features(train_features, features_dict)
encoded_dev_features = encode_features(dev_features, features_dict)
encoded_train_labels = encode_labels(train_tag_sents, labels2i)
encoded_dev_labels = encode_labels(dev_tag_sents, labels2i)
```

```
  0%|          | 0/75827 [00:00<?, ?it/s]
  0%|          | 0/10851 [00:00<?, ?it/s]
Building features set!
100%|██████████| 75827/75827 [00:02<00:00, 33097.46it/s]
Found 208208 features
```

In [55]:
```
# TODO: Play with hyperparameters here.
num_epochs = 5
batch_size = 20
LR=0.1
optimizer = torch.optim.SGD(model.parameters(), LR)
```

In [56]:
```
model = training_loop(
    num_epochs,
    batch_size,
    encoded_train_features,
    encoded_train_labels,
    encoded_dev_features,
    encoded_dev_labels,
    optimizer,
    model,
    labels2i,
    features_dict[PAD_SYMBOL]
)
```

```
Training...
  0%|          | 0/3792 [00:00<?, ?it/s]
/var/folders/31/yzh9j02x7bxd463cl1x0_21h0000gn/T/ipykernel_63767/3584098268.py:55: RuntimeWarning: invalid valu
e encountered in log
  print(f"epoch {i}, loss: {np.log(sum(losses)/len(losses))}")
epoch 0, loss: nan
Dev F1 log tensor([-3.8738])
  0%|          | 0/3792 [00:00<?, ?it/s]
epoch 1, loss: nan
Dev F1 log tensor([-3.8738])
  0%|          | 0/3792 [00:00<?, ?it/s]
epoch 2, loss: nan
Dev F1 log tensor([-3.8738])
  0%|          | 0/3792 [00:00<?, ?it/s]
epoch 3, loss: nan
Dev F1 log tensor([-3.8738])
  0%|          | 0/3792 [00:00<?, ?it/s]
epoch 4, loss: nan
Dev F1 log tensor([-3.8738])
```

Quiz (10 Points)

1. Look at the `NERTagger` class in crf.py

```
a) What does the CRF add to our model that makes it different from the sentiment classifier?
    We accept previous tag as a feature along with the other features has the same reult, we
take the argmax of the sum of the feature scores for each element of the sequence
b) Why is this helpful for NER?
    It takes context into account and recognize forms such as 'has lived', 'is moving'. When a
CRF model makes a prediction, it factors in the impact of neighbouring samples by modelling
the prediction as a graphical model. It assumes that the tag for the present word is dependent
only on the tag of just one previous word
```

2. Why computing F1 here is not straightforward?

Hint: Refer to the class in which Jim went over the evaluation metrics for NER

There are lots of O and you can get 80% accuracy for newspaper article because of that. It looks good but actually not. The evaluation metrics for NER can based on Tag or Entities. You can get 0 accuracy for entities such as it is facility but your model recognize as person or other tag. So it is better use span accuracy, find the prediction in the span or category, do I get subset of span or category right? Need to have different measurement to evaluate the model result. That's why it is not strightforward.

In [ ]: