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

```
!git clone https://github.com/cfiltnlp/HiNER.git
```

```
fatal: destination path 'HiNER' already exists and is not an empty
directory.
```

```
import torch

# 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.

```
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-LOCATION', '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

```
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)
```

```python
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[i] for i in d_ids]
        for d_ids in decoded_pos_ids
    ]
```

```
id of the <unk> token: 2186
```

```
[nltk_data] Downloading package indian to
[nltk_data]     C:\Users\linsh\AppData\Roaming\nltk_data...
[nltk_data]   Package indian is already up-to-date!
```

Feature Engineering Functions (20 Points)

```python
# TODO: Update this function to add more features
#      You can check crf.py for how they are encoded, if interested.

import string
# used from the piazza post regarding the shape of the word
def word_shape(token):
    shape = []
    for char in token:
        if char.isdigit():
            shape.append('d')  # Digit
        elif char in string.punctuation:
            shape.append('p')  # Punctuation
        else:
            shape.append('c')  # Other characters (considered as 
characters)
```

```python
    return ''.join(shape)

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.
        # TODO: Add more features here
        # add neighboring word
        feats.append(f"word={token}")
        if i > 0:
            feats.append(f"prev_word={text[i-1]}")
        else:
            feats.append(f"prev_word={'<s>'}")
        if i < len(text) -1:
            feats.append(f"next_word={text[i+1]}")
        else:
            feats.append(f"prev_word={'</s>'}")

        #part of speech for the word and its neighbors
        feats.append(f"word_pos={get_pos(pos_tagger, [token])[0]}")
        if i > 0:
            feats.append(f"prev_word={get_pos(pos_tagger, text[i-1])
[0]}")
        else:
            feats.append(f"prev_word={'<s>'}")
        if i < len(text) -1:
            feats.append(f"next_word={get_pos(pos_tagger, text[i+1])
[0]}")
        else:
            feats.append(f"prev_word={'</s>'}")

        # word shape
        feats.append(f"word_shape={word_shape(token)}")
        if i > 0:
            feats.append(f"prev_word_shape={word_shape(text[i-1])}")
        else:
            feats.append(f"prev_word={'<s>'}")
        if i < len(text) -1:
            feats.append(f"next_word_shape={word_shape(text[i+1])}")
        else:
            feats.append(f"prev_word={'</s>'}")
```

```
        # We append each feature to a List for the token.
        feature_lists.append(feats)

    return feature_lists

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 sents:
        # Gets a List of Lists of feature strings
        feats.append(make_features(sent))

        # TO DO: Get pos tags
        sent_tags = get_pos(pos_tagger, [sent])[0]

    return feats
```

Finish the training loop. (10 Points)

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

```
from crf import f1_score, predict, PAD_SYMBOL, pad_features,
pad_labels
import random
from tqdm.autonotebook import tqdm

# TODO: Implement the training loop
# HINT: Build upon what we gave you for HW2.
# See cell below for how we call this training loop.
```

```python
def training_loop(
    num_epochs,
    batch_size,
    train_features,
    train_labels,
    dev_features,
    dev_labels,
    optimizer,
    model,
    labels2i,
    pad_feature_idx
):
    # raise NotImplementedError

    # TODO: Zip the train features and labels

    # TODO: Randomize them, while keeping them paired.

    # TODO: Build batches
    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
            emissions = model.make_emissions(features)
            loss = -model.crf_decoder.forward(emissions, labels)

            # 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
        average_loss = sum(losses)/ len(losses)
        print(f"epoch {i}, average loss: {average_loss}")
        # TODO: make dev predictions with the `predict()` function
        dev_predictions = predict(model, dev_features)

        # TODO: Compute F1 score on the dev set and log it.
        dev_f1 = f1_score(dev_predictions, dev_labels,
labels2i[PAD_SYMBOL])
        print(f"Dev F1 {dev_f1}")

    # Return the trained model
    return model
```

```
C:\Users\linsh\AppData\Local\Temp\ipykernel_3696\3381977249.py:3:
TqdmExperimentalWarning: Using `tqdm.autonotebook.tqdm` in notebook
mode. Use `tqdm.tqdm` instead to force console mode (e.g. in jupyter
console)
  from tqdm.autonotebook import tqdm
```

Run the training loop (10 Points)

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

```
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.
```

```python
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)

# TODO: Play with hyperparameters here.
num_epochs = 30
batch_size = 16
LR=0.05
optimizer = torch.optim.SGD(model.parameters(), LR)

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]
)
```

```
Building features set!

100%|██████████████████████████████████████████████████████████████████████| 75827/75827 [00:01<00:00, 43356.15it/s]

Found 224680 features
Training...
```

{"model_id":"9b617ae9874b41168958879b5955dde8","version_major":2,"version_minor":0}

```
C:\Users\linsh\anaconda3\envs\pya3\lib\site-packages\torchcrf\
__init__.py:249: UserWarning: where received a uint8 condition tensor.
This behavior is deprecated and will be removed in a future version of
PyTorch. Use a boolean condition instead. (Triggered internally at C:\
actions-runner\_work\pytorch\pytorch\builder\windows\pytorch\aten\src\
ATen\native\TensorCompare.cpp:519.)
  score = torch.where(mask[i].unsqueeze(1), next_score, score)

epoch 0, average loss: 78.34557910066114
Dev F1 tensor([0.9503])
```

{"model_id":"c7ce2f452d514b5f8f8e03649e8fde98","version_major":2,"version_minor":0}

```
epoch 1, average loss: 39.5550304815236
Dev F1 tensor([0.9563])
```

{"model_id":"8383ed85904e40e68f8f8d1442de6fe6","version_major":2,"version_minor":0}

```
epoch 2, average loss: 31.959528254255464
Dev F1 tensor([0.9594])
```

{"model_id":"074f4fecbd434b8f9aee57048d51eea0","version_major":2,"version_minor":0}

```
epoch 3, average loss: 27.88205841643901
Dev F1 tensor([0.9611])
```

{"model_id":"b364ccf5915941e2baf2dcb227d9e6ed","version_major":2,"version_minor":0}

```
epoch 4, average loss: 25.147945642270116
Dev F1 tensor([0.9629])
```

{"model_id":"eef01310a3fc4a3cb8d654759e8bc472","version_major":2,"version_minor":0}

```
epoch 5, average loss: 23.1127335818005
Dev F1 tensor([0.9640])
```

{"model_id":"e8f3afd6246a4d678468743eb30f5307","version_major":2,"version_minor":0}

```
epoch 6, average loss: 21.50884808568512
Dev F1 tensor([0.9647])
```

{"model_id":"6dcb2bb0b99c4fe59fa12c8a7d853fa3","version_major":2,"version_minor":0}

```
epoch 7, average loss: 20.2003955245521
Dev F1 tensor([0.9653])
```

{"model_id":"a9461cf378094649a33d94a03e1c69cd","version_major":2,"version_minor":0}

```
epoch 8, average loss: 19.105431700758793
Dev F1 tensor([0.9660])
```

{"model_id":"b4fb1b42add54c7988a50ca25a15d6f8","version_major":2,"version_minor":0}

```
epoch 9, average loss: 18.17189932070704
Dev F1 tensor([0.9664])
```

{"model_id":"9b3cccbc1a684236ba4b53bcbdf8f54a","version_major":2,"version_minor":0}

```
epoch 10, average loss: 17.36358060555116
Dev F1 tensor([0.9668])
```

{"model_id":"b89a2e08233b4037822a3f5762edd593","version_major":2,"version_minor":0}

```
epoch 11, average loss: 16.65422584759032
Dev F1 tensor([0.9672])
```

{"model_id":"7adfddfb49dc4b319b41d53e3f5d6516","version_major":2,"version_minor":0}

```
epoch 12, average loss: 16.02508648578628
Dev F1 tensor([0.9674])
```

{"model_id":"1dfeb58767b2412486988af283b29cf4","version_major":2,"version_minor":0}

```
epoch 13, average loss: 15.462388485292845
Dev F1 tensor([0.9677])
```

{"model_id":"6114f653b3604c7faf7c4eef8440d94c","version_major":2,"version_minor":0}

```
epoch 14, average loss: 14.955170060813678
Dev F1 tensor([0.9680])
```

{"model_id":"3df0f5fca6724c0697c15223eb819142","version_major":2,"version_minor":0}

```
epoch 15, average loss: 14.495418439132754
Dev F1 tensor([0.9681])
```

{"model_id":"45d95d0ed79542c39d73f92f7a11ef5e","version_major":2,"version_minor":0}

```
epoch 16, average loss: 14.076497973671442
Dev F1 tensor([0.9683])
```

{"model_id":"9c915c25526c497a963009f9be572100","version_major":2,"version_minor":0}

```
epoch 17, average loss: 13.693202027590466
Dev F1 tensor([0.9685])
```

{"model_id":"08391f65ed524a868b914cde2df8d35a","version_major":2,"version_minor":0}

```
epoch 18, average loss: 13.341027442513639
Dev F1 tensor([0.9687])
```

```
{"model_id":"cab36cd3fbfe495f98cb3d0ac13034ba","version_major":2,"version_minor":0}
```

```
epoch 19, average loss: 13.016398612557584
Dev F1 tensor([0.9689])
```

```
{"model_id":"fa7e228b2a514a44bc9597dbae893ee1","version_major":2,"version_minor":0}
```

```
epoch 20, average loss: 12.716084498795778
Dev F1 tensor([0.9690])
```

```
{"model_id":"3458c9596f4a425fb5de17bf2b9ad908","version_major":2,"version_minor":0}
```

```
epoch 21, average loss: 12.437535104872305
Dev F1 tensor([0.9692])
```

```
{"model_id":"d3e8a2d9e4ca4ce794093afd62ecbb63","version_major":2,"version_minor":0}
```

```
epoch 22, average loss: 12.178712052936795
Dev F1 tensor([0.9693])
```

```
{"model_id":"8a88678be1994220a73a31cb33242dca","version_major":2,"version_minor":0}
```

```
epoch 23, average loss: 11.937234751383464
Dev F1 tensor([0.9694])
```

```
{"model_id":"703602c2c3d34cf782ce5298a5a8e274","version_major":2,"version_minor":0}
```

```
epoch 24, average loss: 11.711721462137085
Dev F1 tensor([0.9697])
```

```
{"model_id":"30b7dd38091d4f1e8ab2c50165eacbde","version_major":2,"version_minor":0}
```

```
epoch 25, average loss: 11.500449477569965
Dev F1 tensor([0.9698])
```

```
{"model_id":"59cdb6fa1d834c019ad136559d08848d","version_major":2,"version_minor":0}
```

```
epoch 26, average loss: 11.30229236747645
Dev F1 tensor([0.9699])
```

```
{"model_id":"1f2355a4f0164b3c86bd768eaa19fba8","version_major":2,"version_minor":0}
```

```
epoch 27, average loss: 11.115710639148825
Dev F1 tensor([0.9700])
```

{"model_id":"4709a92c2e35478eaa34711919ad72a3","version_major":2,"version_minor":0}

```
epoch 28, average loss: 10.940195732277656
Dev F1 tensor([0.9701])
```

{"model_id":"3c629073881948438bbd71ec0ef31331","version_major":2,"version_minor":0}

```
epoch 29, average loss: 10.774374515195436
Dev F1 tensor([0.9703])
```

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?
The CRF adds the ability to model dependencies between tags in the
sequence. In the sentiment classifier, each token's sentiment label is
predicted independently of the other tokens in the sequence. In
contrast, NER involves predicting labels for a sequence of tokens
where the prediction for one token can depend on the labels of the
neighboring tokens.

b) Why is this helpful for NER?
This is helpful for NER because named entities often have structural
patterns and could depend on the context of neighboring words. The CRF
helps the model make coherent predictions for the entire sequence by
considering the dependencies between labels for adjacent tokens.
```

2. Why computing F1 here is not straightforward?

Computing the F1 score for NER is not straightforward. For example, the example given in the textbook with Jane being labeled but not Jane Villanueva. We are likely to experience a similar thing where two words are actually the name that we should have identified; however, we only received partial matching due to the system, resulting in errors. Which will affect the F1 score.