Part A: Parts of Speech Tagging using Hidden Markov Model and Viterbi Algorithm on Hindi Dataset (Total: 40 Points out of 100)

For this assignment, we will implement the Viterbi Decoder using the Forward Algorithm of Hidden Markov Model as explained in class.

Then, we will create an HMM-based PoS Tagger for Hindi language using the annotated Tagset in nltk.indian

You need to first implement the missing code in hmm.py, then run the cells here to get the points

```
from tqdm.autonotebook import tqdm
```

```
C:\Users\linsh\AppData\Local\Temp\ipykernel_13136\987820437.py:1:
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
```

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

%load_ext autoreload
%autoreload 2

from hmm import *
```

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

1st-Order Hidden Markov Model Class:

The hidden markov model class would have the following attributes:

```
1. initial state log-probs vector (pi)
2. state transition log-prob matrix (A)
3. observation log-prob matrix (B)
```

The following methods:

```
1. fit method to count the probabilitis of the training set
2. path probability
3. viterbi decoding algorithm
```

image.png

Task 1: Testing the HMM (20 Points)

```
### DO NOT EDIT ###

# 5 points for the fit test case
# 15 points for the decode test case

# run the funtion that tests the HMM with synthetic parameters!
run_tests()
```

```
Testing the fit function of the HMM
All Test Cases Passed!
Testing the decode function of the HMM
All Test Cases Passed!
Yay! You have a working HMM. Now try creating a pos-tagger using this
class.
```

Task 2: PoS Tagging on Hindi Tagset (20 Points)

For this assignment, we will use the Hindi Tagged Dataset available with nltk.indian

Helper methods to load the dataset is provided in hmm.py

Please go through the functions and explore the dataset

Report the Accuracy for the Dev and Test Sets. You should get something between 65-85%

```
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])
```

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

```
print("No. of unique words in the corpus:", len(observation_dict))
print("No. of tags in the corpus", len(state_dict))
```

```
No. of unique words in the corpus: 2187
No. of tags in the corpus 26
```

```
# Split the dataset into train, validation and development sets

import random
random.seed(42)
from sklearn.model_selection import train_test_split

data_indices = list(range(len(all_observation_ids)))

train_indices, dev_indices = train_test_split(data_indices,
test_size=0.2, random_state=42)

dev_indices, test_indices = train_test_split(dev_indices,
test_size=0.5, random_state=42)

print(len(train_indices), len(dev_indices), len(test_indices))


def get_state_obs(state_ids, obs_ids, indices):
    return [state_ids[i] for i in indices], [obs_ids[i] for i in
indices]


train_state_ids, train_observation_ids = get_state_obs(all_state_ids,
all_observation_ids, train_indices)
dev_state_ids, dev_observation_ids = get_state_obs(all_state_ids,
all_observation_ids, dev_indices)
test_state_ids, test_observation_ids = get_state_obs(all_state_ids,
all_observation_ids, test_indices)

432 54 54

def add_unk_id(observation_ids, unk_id, ratio=0.05):
    """
    make 1% of observations unknown
    """
    for obs in observation_ids:
        for i in range(len(obs)):
            if random.random() < ratio:
                obs[i] = unk_id

add_unk_id(train_observation_ids, observation_dict[UNK_TOKEN])
add_unk_id(dev_observation_ids, observation_dict[UNK_TOKEN])
add_unk_id(test_observation_ids, observation_dict[UNK_TOKEN])

pos_tagger = HMM(len(state_dict), len(observation_dict))
pos_tagger.fit(train_state_ids, train_observation_ids)

assert np.round(np.exp(pos_tagger.pi).sum()) == 1
assert np.round(np.exp(pos_tagger.A).sum()) == len(state_dict)
assert np.round(np.exp(pos_tagger.B).sum()) == len(state_dict)
```

```
print('All Test Cases Passed!')
```

```
All Test Cases Passed!
```

```
def accuracy(my_pos_tagger, observation_ids, true_labels):
    tag_predictions = my_pos_tagger.decode(observation_ids)
    tag_predictions = np.array([t for ts in tag_predictions for t in
ts])
    true_labels_flat = np.array([t for ts in true_labels for t in ts])
    acc = np.sum(tag_predictions ==
true_labels_flat)/len(tag_predictions)
    return acc

print('dev accuracy:', accuracy(pos_tagger, dev_observation_ids,
dev_state_ids))
```

```
dev accuracy: 0.8127659574468085
```

```
print('test accuracy:', accuracy(pos_tagger, test_observation_ids,
test_state_ids))
```

```
test accuracy: 0.7987012987012987
```

```
# Fit a pos_tagger on the entire dataset.
import pickle

full_state_ids = train_state_ids + dev_state_ids + test_state_ids
full_observation_ids = train_observation_ids + dev_observation_ids +
test_state_ids

hindi_pos_tagger = HMM(len(state_dict), len(observation_dict))
hindi_pos_tagger.fit(full_state_ids, full_observation_ids)

pickle.dump(hindi_pos_tagger, open('hindi_pos_tagger.pkl', 'wb'))

### Finally we will use the hindi_pos_tagger as a pre-processing step
for our NER tagger
```

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)
```

```
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)

```
# 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)
```

```
    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.
```

```
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.
```

```
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,"vers
ion_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.

Assignment Title

Programming Assignment (40 points)

The programming assignement will be an implementation of the task described in the assignment

We will make sure you have enough scaffolding to build the code upon where you would only have to implement the interesting parts of the code

Evaluation

The evaluation of the assignment will be done through test scripts that you would need to pass to get the points.

Written Assignment (60 Points)

Written assignment tests the understanding of the student for the assignment's task. We have split the writing into sections. You will need to write 1-2 paragraphs describing the sections. Please be concise.

In your own words, describe what the task is (20 points)

The first part of the assignment is a part of speech tagging using the Hidden Markov Model and the Viterbi Algorithm. POS tagging aims to assign the appropriate part of speech to each word in a given text or sentence. For example, given the context of the word "back," it could be ADJ (The back door), NOUN (On my back), ADV (Win the voters back), and VERB (Promised to back the bill). In this context, we are assigning part of speech to a Hindi dataset, tagging each word to the appropriate part of speech. However, given my lack of knowledge of Hindi, I could not provide an example. Lastly, part of speech tagging is crucial to semantic analysis, parsing, information retrieval, and sentiment analysis, just to name a few.

The second task is named entity recognition with the conditional random field (CRF). NER aims to identify and classify named entities in text into predefined categories, such as names of persons, organizations, locations, etc. For example, we could classify (PER)(Jane Villanueva) of (ORG)(United), a united of (ORG)(United Airlines Holding), said the fare applies to the (LOC)(Chicago) route. In this context, we classify them into the appropriate categories in the Hindi dataset. Lastly, NER is one of the first steps for an NLP application; it could be useful in various applications, such as information retrieval, extraction, text summarization, etc.

Describe your method for the task (10 points)

For the first task, I completed the hmm.py file (fit() and decode()) based on the comments and hints that were provided. Once the code was complete, I ran the Jupyter Notebook to ensure everything passed. Therefore, no additional implementation and design details are worth discussing in Part 1.

For the second task, I completed the "training loop" section by using what was used in HW2. For the feature engineering, I used Figure 8.15 as a reference. However, we could not complete all eight features. For example, without the knowledge of Hindi, I am unsure what to use for a gazetteer. Additionally, training and running the code is extremely long, depending on the number of features implemented. Therefore, I decided to implement a few suggested features to balance time and the number of features.

The first feature I implement is identifying w_i and its surrounding words. When the word is at the beginning, I assign ¿s> as the previous word. When the word is at the end, I assign ¿/s> as the word after. The next feature I implemented is part-of-speech of w_i for the word and its surrounding words. I used the same ¿s> and ¿/s> for the beginning and ending token. The last feature that I implemented is the word shape. However, upper and lower case does not matter since we work in Hindi. Therefore, I added a few points: if the character is a digit, punctuation, or just a character. Given the three implementations, the run time is around 1 hour to 2 hours per sample run. Therefore, I decided to stick with the three features to run enough experiments.

Then to tune the parameters, I conducted a few sample runs by tuning the num_epochs, batch_size, and the LR. Ultimately, I decided the default value provided was sufficient for the run. Although running the experiment with a higher number of num_epochs will yield better loss and F1 score, the runtime is nearly double to get a small improvement (see experiment results); therefore, I decided to stick with the default value of 30 (num_epochs), 16 (batch_size), and 0.05 (LR).

Experiment Results (10 points)

num_epochs	batch_size	LR	Loss	F1
30	16	0.05	10.769497667787448	0.9701
60	16	0.05	8.199515562319052	0.9748
50	**20**	0.05	10.951339900745118	0.9701
30	**32**	0.05	22.078188866402026	0.9718
60	**32**	0.05	16.525589135729312	0.9728
30	16	**0.1**	9.37492883748646	0.9679
30	16	**0.01**	19.888798390963913	0.9672

The above is a selective subset that represents the different experiments conducted. At one time, I even randomized these hyperparameters to test them out. These results show that the initial default values yield a pretty good F1 score. Modifying the hyperparameters could significantly increase the run time while having little to no difference in the F1 score.

In the end, the final run of 30 (num_epochs), 16 (batch_size), and 0.05 (LR) has an average loss of 10.774374515195436 and an F1 score of 0.9703.

Discussion (20 points)

The key takeaway from the assignment is exploring and implementing two fundamental natural language process tasks: Part-of-Speech tagging using Hidden Markov Models and Named Entity Recognition using Conditional Random Fields. The two tasks are crucial and fundamental to various NLP tasks. Implementing them gave me more insights into the models.

The methods are working pretty well since they explore the sentence structure. They are considering neighboring tokens, which is important in tasks like POS tagging and NER, where the surrounding words will hint at the final solution. Additionally, even without any modification to the features used, it had an above 0.9 F1 score, indicating the methods used were well-suited for the task.

Although the F1 score indicated a pretty well-trained model, several shortcomings exist. For example, I could not implement all the suggested features due to the limited resources. Additionally, with no knowledge of Hindi as the language, the methods depend on the neighboring words, but what if the dependencies are not limited to just the words nearby? Lastly, due to the limited resources, I could not increase some of the hyperparameters and run more experiments from a larger range.

Most of the issues I mentioned above could be resolved by having more time, running more experiments, and having more resources. I would identify a reasonable gazetteer to help with the geographic identifications. I would include all the features suggested in Figure 8.15. I would try other methods to test against the current method of choice.

I enjoyed this assignment, as it gave me more insights into the two tasks. At the same time, I would love to have a way to validate the results myself. Given the discussion about how the F1 score is not straightforward, I wanted to examine the results and have another way to analyze what the model is telling me. However, given my lack of knowledge of Hindi, I can only use the F1 score from the model. That said, I still enjoyed the assignment and learning about the model; I just hoped I could have another way to validate it (or not).