Assignment 2

Pin-Jie Lin

pili00001@stud.uni-saarland.de
https://github.com/pjlintw/CL20/tree/main/assignment2

1 Implementation of POS tagging with HMMs

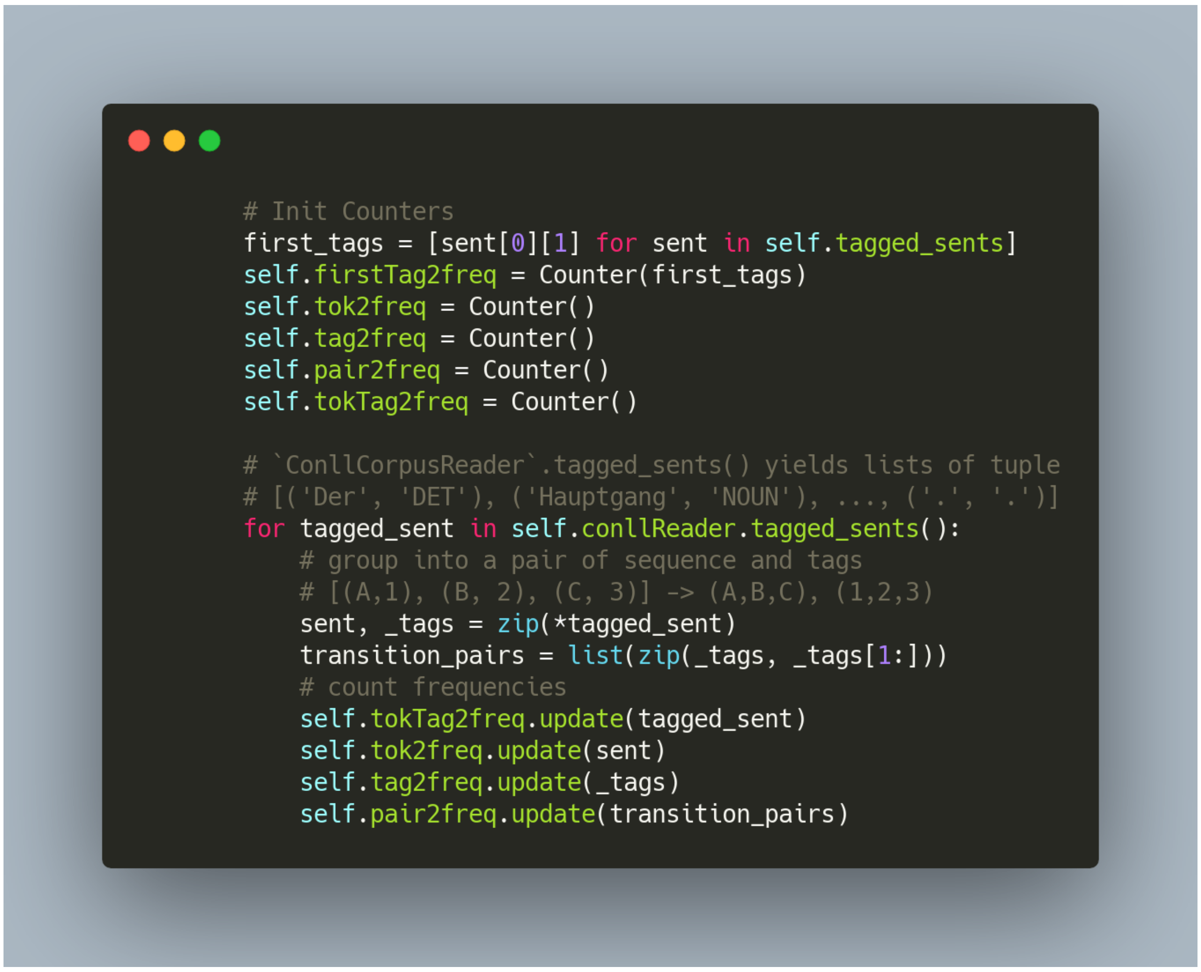
We implement all necessary functions including unknown words dealing under a class `HiddenMarkovTagger` , which is implemented as a bigram-based HMM tagger and can be tranined and evaluated on CoNLL datasets. The tagger estimates an optimal sequence of tags by using Viterbi algorithm `_viterbi_searching()` . The model uses de-utb/de-train.tt for traning, evaluating its performance on file `de-utb/de-test.t` . Modify the file path if you are not using the same dataset.

1.1 Parameters and Viterbi of HiddenMarkovTagger

HMM parameters. The main parameters of HMM tagger consist of three probabilities, namly **initial probability**, **transition probability** and **emission probability**. Since we estimate a bigram-based tagger, we can compute the probabilities with frequencies of tag, word and the transition. The abstract formula of probabilities are:

```
initital probability  : num_state_i_at_idx_0 / num_sents
transition probability: num_state_i2j        / num_state
emission probability  : num_state_token_ij   / num_state_i
```

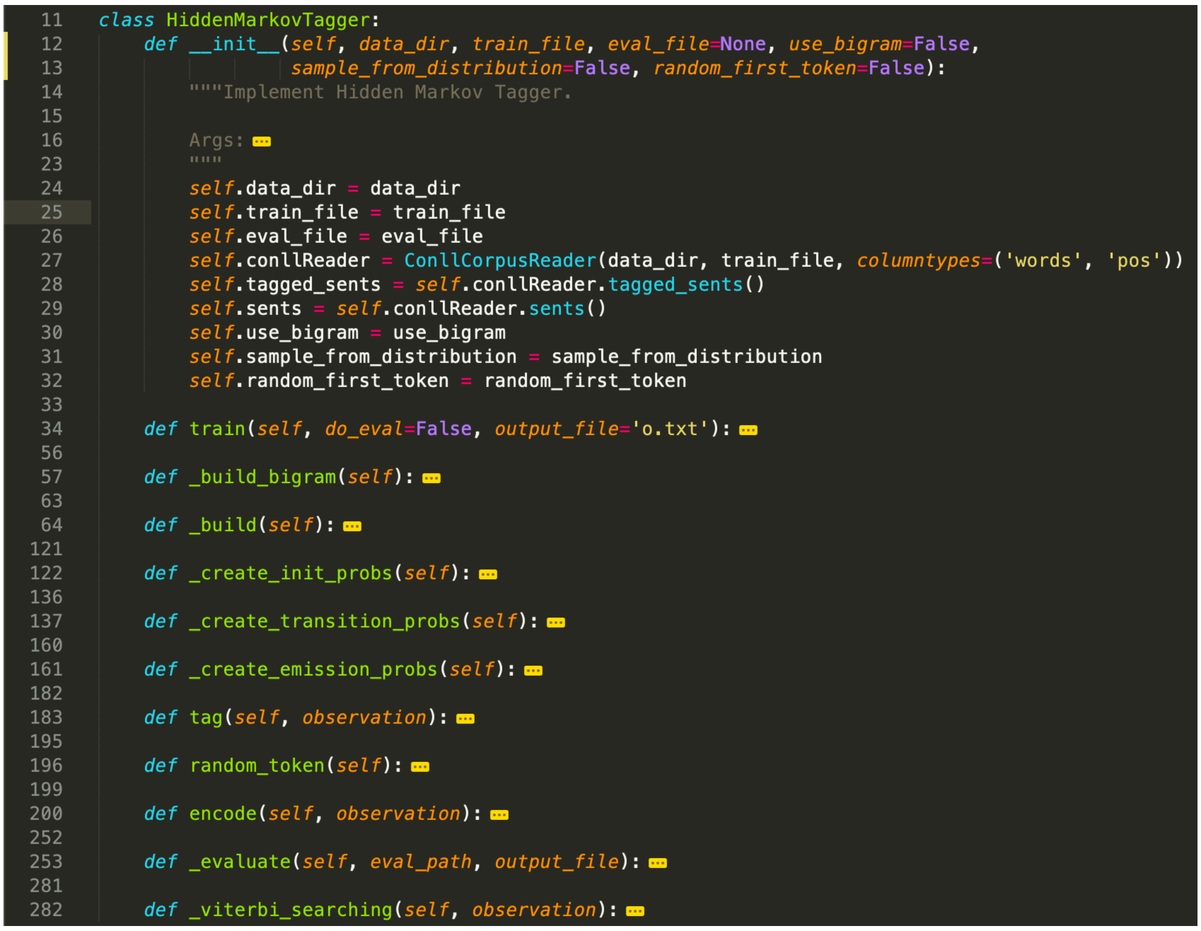
We compute the frequencies of the training set first. Then creating three probabilities sparately.



```python
# Init Counters
first_tags = [sent[0][1] for sent in self.tagged_sents]
self.firstTag2freq = Counter(first_tags)
self.tok2freq = Counter()
self.tag2freq = Counter()
self.pair2freq = Counter()
self.tokTag2freq = Counter()

# `ConllCorpusReader`.tagged_sents() yields lists of tuple
# [('Der', 'DET'), ('Hauptgang', 'NOUN'), ..., ('.', '.')]
for tagged_sent in self.conllReader.tagged_sents():
    # group into a pair of sequence and tags
    # [(A,1), (B, 2), (C, 3)] -> (A,B,C), (1,2,3)
    sent, _tags = zip(*tagged_sent)
    transition_pairs = list(zip(_tags, _tags[1:]))
    # count frequencies
    self.tokTag2freq.update(tagged_sent)
    self.tok2freq.update(sent)
    self.tag2freq.update(_tags)
    self.pair2freq.update(transition_pairs)
```

Figure 1. Creating Counter to record the frequencies of the token, tag, token at first position, the transition and the emission.

Structure of HiddenMarkovTagger. The class has one main method `.train()` for both training and evaluting. The steps for using the class are only constructing the class and calling its `.train()` method. We can run the evaluation by passing `True` to `.train(do_eval=True, output_file='o.txt')` parameter. The method will automatically evaluate the performance of estimated probabilities on the test file given from `eval_file`. When evaluating the HMM tagger, `_viterbi_searching` will estimate probabilities for decoding an optimal tag sequence given a observation.



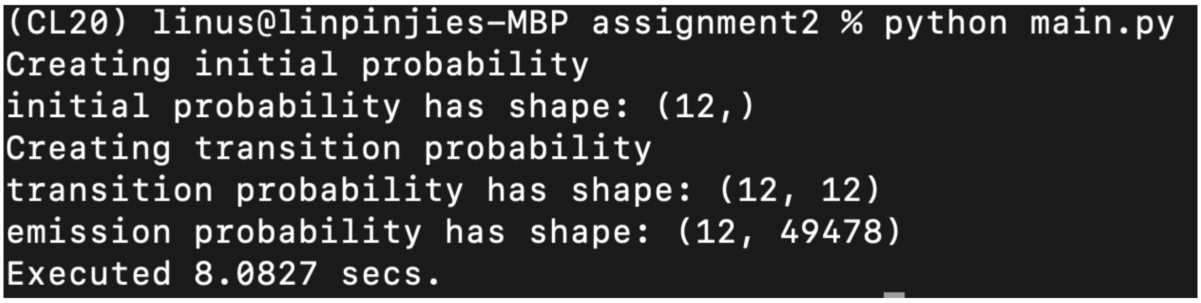
```python
class HiddenMarkovTagger:
    def __init__(self, data_dir, train_file, eval_file=None, use_bigram=False,
                 sample_from_distribution=False, random_first_token=False):
        """Implement Hidden Markov Tagger.

        Args: ...
        """
        self.data_dir = data_dir
        self.train_file = train_file
        self.eval_file = eval_file
        self.conllReader = ConllCorpusReader(data_dir, train_file, columntypes=('words', 'pos'))
        self.tagged_sents = self.conllReader.tagged_sents()
        self.sents = self.conllReader.sents()
        self.use_bigram = use_bigram
        self.sample_from_distribution = sample_from_distribution
        self.random_first_token = random_first_token

    def train(self, do_eval=False, output_file='o.txt'): ...

    def _build_bigram(self): ...

    def _build(self): ...

    def _create_init_probs(self): ...

    def _create_transition_probs(self): ...

    def _create_emission_probs(self): ...

    def tag(self, observation): ...

    def random_token(self): ...

    def encode(self, observation): ...

    def _evaluate(self, eval_path, output_file): ...

    def _viterbi_searching(self, observation): ...
```

Figure 2. The structure of `HiddenMarkovTagger`

2. Running the HMM tagger

To train the tagger and evaluate on test file. Simply execute `main.py` in command line.

```
python main.py
```



```
(CL20) linus@linpinjies-MBP assignment2 % python main.py
Creating initial probability
initial probability has shape: (12,)
Creating transition probability
transition probability has shape: (12, 12)
emission probability has shape: (12, 49478)
Executed 8.0827 secs.
```

Figure 3. It shows the shapes of probability matrices and the execution time for training `HiddenMarkovTagger`.

3. Dealing with unknown words (extra credit)

In our implementation, we design 4 mechanisms to deal the missing token in vocabulary while inference.

Assumption. Since we've learnd to produce the next token using N-gram model by given most recent context. We made an assumption that **the missing tokens must follow certain previous words**. Therefore, we replace the unknown word with a generated token from bigram model trained on the training set, which make sure the generated token is in the vocabulary.

Model design. Based on the assumption, we design 2 models using generated token replacing method. The **HMM(bigram)**, which is a baseline, uses bigram model to replace the unknown word by predicting the missing one in vocabulary given one previous word. In order to produce the missing token at first position, we add a start of sentence token `[SOS]` to vocabulary and to each sentence in the training corpus for bigram model. During inference, whenever encountering a unknown word in the beginning of sentence, we fit the `[SOS]` to the trained ngram model to generate first token. All the other unknown words in the sentence will be replaced by a generated token given one previous token. If the previous word is unknown, the ngram will use its the privous generated token as input.

Since the generated token for the first position have always the same previous context `[SOS]`. We want the first generated token to be generated in more random way. The **HMM(bigram+randomFirstToken)** tries to improve the unknown word at first position by random sampling first words from all first words in training sets.

After the two models, we considers a more general way for dealing unknown word. The third one **HMM(sampleDistribution)** simply replaces all unknown word by **sampling a token from word distribution** of the training sets. Finally, **HMM(empirical)** follows empirical rule of unknown word, which usually are a `NOUN`. We replace all unknown words by a word whose tag is NOUN.

Configurations To execute various HMM taggers, we have to adjust the parameters of HiddenMarkovTagger (Line 335 in `main.py`). There are three parameters for controlling which mechanisums will be used, `use_bigram`, `sample_from_distribution` and `random_first_token`. To use different functions for unknown word, pass `boolean` to the parameters of the class like the table bellow. Notice that: random_first_token will be used when use_bigram is True.

```python
# Line 330 in main.py
hmm_tagger = HiddenMarkovTagger(data_dir=DATADIR,
                                train_file=train_file,
                                eval_file=test_file,
                                use_bigram=False,
                                sample_from_distribution=Fa
lse,
                                random_first_token=False)
```

Experimental Result. We made the smiliar previous context assumption for first two models **HMM(bigram)** and **HMM(bigram+randomFirstToken)**. The bigram model for generating unknown token gives us excellent score for accuracy. When we apply `randomFirstToken` for the first unknown word on top of **HMM(bigram)**. It can get one more points . We are gussing that the probability of generating the first token from training corpus and testing corpos are not exactly smilar. Therefore, random sampling first token can perform better.

We also try a general way to replace the unknown word. The method of sampling from distribution get 3 more points than baseline. But the execution is pretty slow. We then apply the prior knowledge of unknown words for the mehtod by simply replacing every unknown word with a token `Kosten` , whose tag is `NOUN` . It shown that the **HMM(empirical)** is the fastest and outperform the others.

models	use bigram	random first token	sample from distribution	speed (sec.)	accuracy
HMM(bigram)	`True`	`False`	`False`	slow (8.7685)	0.8411
HMM(bigram+randomFirstToken)	`True`	`True`	`False`	slow (9.3351)	0.8513
HMM(sampleDistribution)	`False`	`False`	`True`	extremly slow (18.4848)	0.8707
HMM(empirical)	`False`	`False`	`False`	**fast (6.8549)**	**0.8714**

Figure 4. Various `HiddenMarkovTagger` and its performance.