Tutorial: Statistical Language Processing (CL III)
Summer 2023

Assignment 3: Due Monday, June 19th, 2023 - 17:00

Submission

See the general rules which apply to all assignments. Submit the following files to Moodle (only 1 submission per group):

- tagger.py
- baseline.py
- hmm.py

Sequence Labeling

In this assignment you will get practically acquainted with Hidden Markov Models (HMMs). In particular, you will tackle the decoding task (i.e. finding the best label sequence given an observation sequence) using a first-order HMM. Chapter 8 and Appendix A of J&M provide a detailed insight into HMMs and their use in sequence labeling. To provide examples, the authors discuss the problems of POS-tagging and Named Entity Recognition (NER).

You will implement two POS taggers: a baseline implementation that uses a very simple algorithm, and an HMM implementation. The baseline implementation is needed for comparison. An F1-score in isolation does not tell us much. We need to compare it to the score of a different implementation that was trained and evaluated on the same data. Of course, the goal is get a better score with the HMM than the baseline implementation.

Steps:

1. Train the baseline model by creating a {word: POS} dictionary, where the POS assigned to each word is simply the most frequent POS for word in the training data.
2. Evaluate the baseline model on the testing data.
3. Train the HMM by creating the initial probability vector (pi), the transition matrix (A), and the emission matrix (B), as well as index maps into these matrices. Several ideas from n-gram modelling are used in HMMs as well:
    - Smoothing: Add-k smoothing will be applied to the matrices to avoid probabilities of 0.
    - Sentence markers: EOS markers (and artificial END POS tags) will be added to capture information about sentence endings. BOS markers are not necessary, since we calculate the beginning of sentence probabilities separately in pi.
    - Handling OOV words in the test data with the <UNK> token
4. Evaluate the HMM on the test data.

Training and evaluation takes roughly 8 seconds (baseline), 15 seconds (HMM) on my machine.

Dataset

We will use the UD English Web Treebank data. Download the train and test data (we won't use the dev data) into the **Data** directory of the starter code.

The files are in the CoNLL-U format, which is a simple and widely-used dependency treebank format.

The basic, essential information about CoNNL-U you need to complete this assignment:

- Lines starting with "#" are comments

- Each token in a sentence is on a separate line which contains exactly 10 tab-separated columns (word id, word form, lemma, upos, xpos, feats, head, deprel, deps, misc).
- The character "_" in a column indicates that no information is available for that column, or that the column is not relevant to the token.
- Sentences are separated by one blank line.
- We only need the word **form** and **upos** (UD POS tags) columns for this assignment. (The xpos tags are optional, fine-grained, language-specific POS tags.)
- Some words are split into several tokens (multiword tokens). For example, "don't" is split into "do" and "n't".
- There are a small number of cases where the word **form** of a token is None. These are tokens that were not contained in the original text, but were added by the treebank annotators to make the sentence complete/grammatical.

Your taggers will use the word **form** and **upos** information for training.

1 Starter Code

The starter code consists of 3 files:

- tagger.py

  Contains the class Tagger, which is the parent class of HMM (in hmm.py) and Baseline (in baseline.py). The Tagger class implements methods that are common to both child classes: load_data(), get_mapping(), replace_oov(), evaluate_model(). Training data (sentence tokens and their corresponding POS tags, vocab, tagset) and testing data (sentence tokens and their corresponding POS tags) are stored in Tagger class variables.

- baseline.py

  Contains the implementation of the Baseline class, which assigns the POS tag of each word in the vocabulary the tag that is most frequent for that word in the training data.

- hmm.py

  Contains the implementation of the HMM class, which builds a set of matrices from the training data, and uses the viterbi algorithm for decoding (making predictions).

2 Tagger class (Total: 10 pts)

2.1 *load_data()* (4 pt)

Implement the function load_data() according to the instructions in the code.

It is convenient to use the pyconll package to read conllu files. Its use is similar to that of spaCy, except that it does not annotate data, rather it reads annotated data from a file.

Please use **pyconll.iter_from_file()**, which loads one sentence at a time into memory (rather than the entire corpus).

Multiword tokens, such as "don't" should be skipped. The example below shows the first 8 columns of a sentence with a multi-word token. The token information is contained in the token sub-parts, not in the multi-word token itself. See the pyconll documentation to find out how to retrieve token attributes.

Tokens whose word **form** is None should also be skipped.

```
1   They    they    PRON  PRP Case=Nom|Number=Plur|Person=3|PronType=Prs             4   nsubj   ...
2-3 don't   _       _     _   _                                                     _   _       ...
2   do      do      AUX   VBP Mood=Ind|Number=Plur|Person=3|Tense=Pres|VerbForm=Fin  4   aux     ...
3   n't     not     PART  RB  _                                                     4   advmod  ...
4   believe believe VERB  VB  VerbForm=Inf                                          0   root    ...
5   him     he      PRON  PRP Case=Acc|Gender=Masc|Number=Sing|Person=3|PronType=Prs 4   obj     ...
6   .       .       PUNCT .   _                                                     4   punct   ...
```

2.2 *get_mapping()* (2 pt)

Implement the get_mapping() function, which takes a list[str] and returns a dictionary mapping {idx:str}, or {str:idx} if reverse=True. These maps link column or row names with indices, and are needed to locate elements in the matrix components.

2.3 *replace_oov()* (2 pt)

Implement the replace_oov() function which returns a list[list[str]]) where all words in self.test_sents which are not in self.vocab are replaced with the UNK token.

2.4 *evaluate_model()* (2 pt)

Implement the evaluate_model() method according to the instructions in the code.

3 Baseline class (Total: 10 pts)

3.1 *train_model()* (6 pt)

Implement the train_model() method according to the instructions in the code.

3.2 *decode()* (2 pt)

Implement the decode() method according to the instructions in the code.

3.3 *main()* (2 pt)

Implement the main() method according to the instructions in the code.

4 HMM class (Total: 25 pts)

In order to use the Viterbi algorithm for the decoding task, we have to define the components of the model. Use the loaded training data and Maximum Likelihood Estimate (MLE) to calculate the probability matrices.

4.1 *initial_probs()* (2 pt)

Implement the initial_probs() function which calculates the initial probability distribution over states (tags), i.e. the probabilities of a sentence starting with each tag. Apply Add-k smoothing.

4.2 *transition_matrix()* (4 pts)

Implement transition_matrix(), which calculates the transition probability matrix, i.e. the probabilities of one tag following another in a sentence. Apply Add-k smoothing.

4.3 *emission_matrix()* (4 pts)

Implement the emission_matrix() function which calculates the emission probability matrix, i.e. the probabilities of a particular observation (word) being generated from a particular state (POS tag). Use the class variable maps {tag:idx} map for row indexing, and the {word:idx} map for column indexing. Apply Add-k smoothing.

4.4 *train_model()* (4 pts)

Implement the train_model() method as described in the code.

4.5 *decode()* (viterbi algorithm) (10 points)

Use the Viterbi algorithm to predict the POS tags for the input sentence.

Implement the decode() function that takes a sentence (as list[str] of tokens), and returns the predicted POS tags, i.e. the POS-tag sequence that is assigned the highest score according to our HMM, as well as the probability of the prediction.

Implement the Viterbi algorithm according to the pseudo-code below, which follows the description in J&M, but expressed in a more python-like style, the model is not passed in, rather its components are stored in class variables, and we do not return the best path probability here.

```
function Viterbi(observation of len T)
  # observation is a list[str] of sentence tokens of length T
  # A is the transition matrix (probs of one POS following another)
  # B is the emission matrix (probs of word given POS)
  # A_map is the {idx: tag} map for access to A
  # B_map is the {word:idx} map for access to B columns

  create path probability matrix viterbi shape (N, T)  # N = the number of states (POS tags)
  create backpointer matrix backpointers shape (N, T)
  for each state s from 0 to N-1 do:                   # initialization step
    viterbi[s][0] <- pi[s] * B[s][B_map[sent[0]]
    backpointers[s][0] <- 0

  for each time step t from 1 to T-1 do:               # recursion step
    for each state s from 0 to N-1 do:
      viterbi[s][t] <- max(viterbi[s_prime][t-1] * A[s_prime][s] * B[s][B_map[sent[t]])
      backpointers[s][t]<-argmax(viterbi[s_prime][t-1] * A[s_prime][s] * B[s][B_map[sent[t]])

  # best_path_prob <- max(viterbi[s][T])             # not returned in this impl
  best_path_pointer <- argmax(viterbi[s][T])       # termination step

  best_path <- the path starting at state best_path_pointer,
      that follows backpointers[] to states back in time

  prediction <- POS ID's in best_path, converted to POS tags
  prediction <- prediction, reversed

  return prediction
```

As an example, your function should work as follows:

```
viterbi(["Google", "is", "a", "nice", "search", "engine", ".", "</s>"], HMM=(pi, A, B, A_map, B_map))
>>> ["PRON", "AUX", "DET", "ADJ", "NOUN", "NOUN", "PUNCT", "END"], 1.943357926500911e-23
```

4.6 *main()* (1 pt)

Implement the main() method according to the instructions in the code.

If your implementations are correct, you should get an F1-score of .79 using the baseline implementation, and .76 using the HMM with k=1.0. However, you should be able to beat the baseline by adjusting the smoothing factor k.

Although it's not a requirement of this assignment, you are encouraged to try training your tagger with one of the other Universal Dependency treebanks. There are hundreds of UD treebanks, in many languages, and you should be able to use any of the others without changing your code.