Classes and methods

Conll.py

A 'Conll' class is implemented:

- **Conll:**
  __init__(self,filename): a Conll object is instantiated with the parameter *filename*. The file has to be a Conll file with two columns: the first one containing the words of the corpus, the second one containing the part-of-speech tags. Sentences are separated by a newline.

  The following attributes are initialized:
  text is initialized to the nltk conll corpus reader of the file.
  tagged_sents is initialized to a list of lists of tuples (word,tag) for each sentence, by calling the function tagged_sents() on the attribute *text*.
  tagged_words is initialized to a list of tuples (word,tag), by calling the function tagged_words() on the attribute *text*.
  tags is initialized to a tuple of unique tags in the text: it is created out the set of the list of all tags in the text.
  vocabulary is initialized to a tuple of unique words in the text: it is created out the set of the list of all words in the text.

  pdist_initial_tags(self): returns an MLEProbDist object: for each sentence in *self.tagged_sents*, it appends the first tag in a list initial_tags. A FreqDist object is created out of this list, and converted to an MLEProbDist object.
  #NOTE: The probabilities in an MLEProbDist always sum to one (the class attribute SUM_TO_ONE is set to True).

  cpdist_tags_bigrams(self): returns a ConditionalProbDist of the tags, given the preceding tag as condition P(tag|tag-1): for each sentence in *self.tagged_sents*, it appends the tags in a list sent_tags, creates a list of bigrams and adds it to the initially empty list tags_bigrams. We work on each single sentence, because a simple list of all the tags in the text would not distinguish the initial probabilities.
  A ConditionalFreqDist object is created from tags_bigrams, and converted to a ConditionalProbDist object with an MLEProbDist.
  #NOTE: Using Laplace for the transition probabilities between tags wouldn't make much sense, since it's unlikely that unseen two-tags sequences would be even grammatical.
  ##NOTE: The ConditionalProbDist object doesn't display a SUM_TO_ONE attribute, BUT the MLEProbDist that it contains for each tag do.

  cpdist_tags_words(self,distribution): returns a ConditionalProbDist of the words, given the assigned tag as condition (P(word|tag)).
  A ConditionalFreqDist object is created from the list of tuples (tag,word) from *self.tagged_words*.
  The distribution parameter to be specified is either 'MLE' or 'Laplace'. Depending on the chosen distribution, the ConditionalFreqDist is converted

to a ConditionalProbDist with an MLEProbDist or with a LaplaceProbDist.
#NOTE: The ConditionalProbDist object doesn't display a SUM_TO_ONE attribute, BUT:
- if we used Maximum Likelihood Estimation, the MLEProbDist that it contains for each tag does.
- if we smoothed the probabilities with Laplace, the LaplaceProbDist for each tag doesn't display a SUM_TO_ONE attribute, since a probability of 1/bins is given also for unseen events. We can check, though, that the probabilities over our known vocabulary and tags still sum to one. The sum of the probabilities over each tag will be printed - approximation due to decimal floating-point numbers.

HMM.py

An 'HMM' class is implemented:

- **HMM:**
  __init__(self,Q,O,aij,a0i,b): an HMM object is instantiated with the parameters *Q, O, aij, a0i, b*.
  Q is a tuple of states.
  O is a tuple of possible observations.
  aij is a QxQ matrix that stores the transition probabilities.
  a0j is a list that stores the initial probabilities for each state in Q.
  b is a QxO matrix that stores the emission probabilities.

  An attribute is initialized for each parameter:
  Q
  O
  aij
  a0i
  b

  Backpointers(self,observation): an attribute bp is initialized to a Qx(O-1) matrix.
  #NOTE: We don't need to store the indexes for the last round of Viterbi values.

  Viterbi_p(self,observation): returns a list of the last recursive call of Viterbi_p. The parameter *observation* is a list of observations.
  A list V_t is instantiated.
  If the observation list only contains one observation, for each state in Q we compute the V value by multiplying its corresponding a0i value and b value on the observation. If the observation is unseen (not in O), *self.b[i][self.O.index(observation[0])* will raise a ValueError. In this case, V is instantiated to a0i value only (we set b=1). The V_t list is filled with all the V scores, and returned.
  If the observation list contains more than one observation, a list V_t_minus_1 is instantiated to a call to Viterbi_p() on the observation

list deprived of the last element. For each state in Q we compute the V value by multiplying the V values of the previous iteration (from V_t_minus_1) with the transition value of the state to all other possible states, and the emission value from the state to the last observation in the observation list. If the observation is unseen (not in O), *self.b[j][self.O.index(observation[0])* will raise a ValueError. In this case, we set b=1. The maximum V for each state is appended to the V_t list, and the index of the corresponding i state is stored in the backpointers matrix. An attribute V_t is initialized to the V_t list and returned.
#NOTE: I didn't use the log of the probabilities, since there are 0 probabilities when using the MLE distribution, and log(0) is undefined.

BestSequence(self,observation): returns a list with the best sequence of states for the given observation, according to our HMM. The parameter observation is a list of observations.
Self.Backpointers is called on the observation.
Self.Viterbi_p is called on the observation.
A best_sequence attribute is initialized to an empty list. The index of the maximum V value is retrieved from *self.V_t*, and the corresponding state is appended to the best_sequence list – which will correspond to last timestep, therefore the last observation.
For each previous timestep, we retrieve the index of the state tag that corresponds to the current state tag in the backpointers matrix. We add the corresponding state tag and insert it at the beginning of the best_sequence list. The list is returned.

Two functions are defined outside the class:
pdist_to_list(pdist,rows): transforms a ProbDist object in a list, given the list of states the rows should correspond to. In practice, the function was implemented to transform the ProbDist of initial probabilities into a list, where the indexes would correspond to the list of tags in our hmm.
cpdist_to_matrix(cpdist,rows,colums): transforms a ConditionalProbDist object in a matrix, given the list of states the rows should correspond to, and the list of symbols the columns should correspond to. In practice, the function was implemented to transform the ConditionalProbDist of transition and emission probabilities into matrices, where the indexes of the rows would correspond to the list of tags in our hmm, the indexes of the columns would correspond to the list of symbols (tags or vocabulary).
#NOTE the functions are implemented to make the probabilities readable to the HMM class.

POS_Tagger.py

A 'POS_Tagger' class is implemented:

- **POS_Tagger:**
  __init__(self,hmm,filename,sentences): a POS_Tagger object is instantiated with the parameters *hmm*, *filename* and *sentences*.
  hmm is an HMM object.

filename is the string of the name of the file to be tagges.
sentences is a list of lists, each one containing the split words of a sentence.

An attribute is initialized for each parameter:
hmm
filename
sentences

Generate_POS(self): writes a 'filename_tagged.tt' file to the current directory in the Conll format.
For each sentence in self.sentences, the BestSequence method is called with our hmm. Each word with the corresponding tag is written to the file. Sentences are separated by a newline.

How to run the program and execution

The execution of the program follows these steps:

1) Open POS_Tagger.py in python3.

2) You will be asked to choose a probability distribution among 'MLE' and 'Laplace', with which you would like to calculate your emission probabilities.

3) The files "de-test.t" and "de-train.tt" should be in your current directory. If not, they will be automatically downloaded.

4) A Conll object is instantiated on the training corpus.

5) An HMM object is instantiated from the parameters of the training corpus:
   - Q corresponds to the tags.
   - O corresponds to the vocabulary of the corpus.
   - aij corresponds to the transition probabilities: a ConditionalProbDist object is instantiated from the Conll object with the cpdist_tags_bigrams() method, and then transformed into a matrix.
   - a0i corresponds to the initial probabilities: a ProbDist object is instantiated from the Conll object with the pdist_initial_tags() method, and then transformed into a list.
   - b corresponds to the emission probabilities: a ProbDist object is instantiated from the Conll object with the cpdist_tags_words() method, and then transformed into a matrix.

#NOTE The modality of instantiation of b will depend on the initially chosen distribution.

5) A list of lists of split sentences is created from the test file.

6) A POS_Tagger object is instantiated with the hmm, the name of the file, and the split sentences of the test file.

7) The tagged file is written.