VINCENT MICHAEL SUTANTO - 2111434
vincent.michael_sutanto.vg6@is.naist.jp

Q. Traditional sequence labelling and parsing techniques were applied in various fields of NLP, now also they can be useful with deep learning. In this assignment, we will learn traditional techniques through part-of-speech tagging and dependency parsing. Your missions are:

- - Create your own model with handcraft features to predict the correct class of a dataset. It can be: Hidden Markov Model(HMM), Conditional Random Fields(CRF), etc (DO NOT USE DEEP LEARNING method).

- Write your short report: Write a short explanation about your program and show us your code. Also, show your accuracy on the test set.

Training file: https://github.com/neubig/nlptutorial/blob/master/data/wiki-en-train.norm_pos

Test file: https://github.com/neubig/nlptutorial/blob/master/data/wiki-en-test.norm_pos

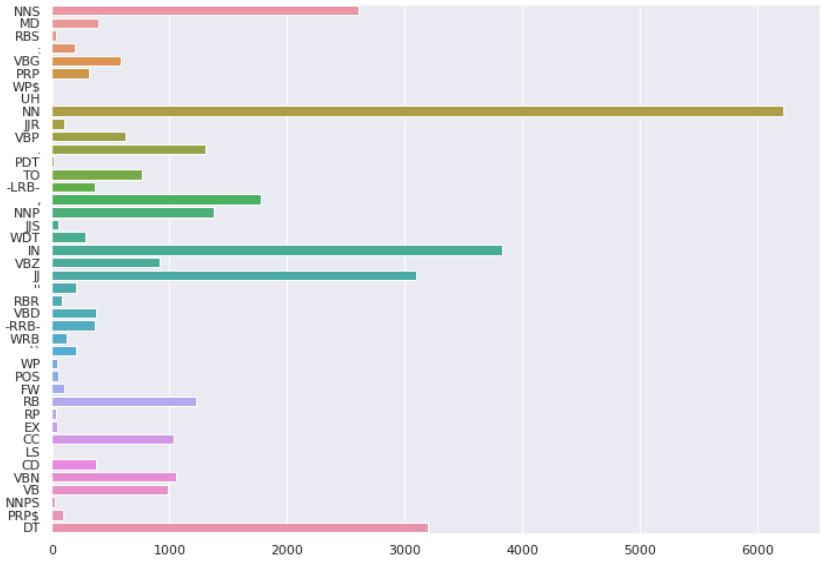
Abstract

In this task, ***Hidden Markov Model*** *with* ***Viterbi Algorithm*** *is used to do part-of-speech (POS)tagging, a common task in Natural Language Processing. The final result produced an* ***accuracy score of 90.62%****. I also did a few more steps: 1. visualize the Viterbi algorithm during training; 2. Displays word token and prediction results pairs.*

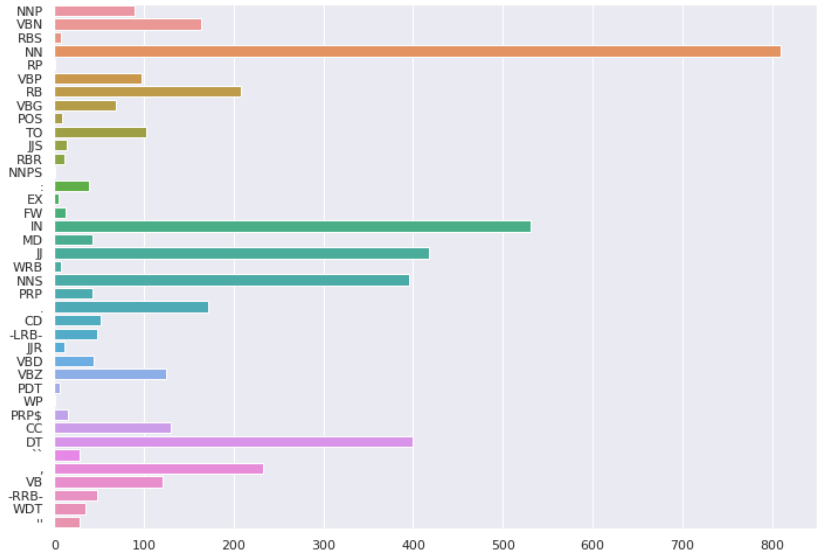
**please see the notebook for the code, because I think it would be too long to explain all the code in this report*

Data Analysis

I did some data analysis regarding the training data used. The training data consists of 42 part-of-speech tags, which consists of 'NNP', 'VBN', 'RBS', 'NN', 'RP', 'VBP', 'VBG', 'RB', 'POS', 'TO'. ', 'JJS', 'RBR', 'NNPS', ':', 'EX', 'FW', 'IN', 'MD', 'JJ', 'PRP', 'NNS', 'WRB', '.', 'CD', '-LRB-', 'JJR', 'VBD', 'VBZ', 'PDT', 'WP$', 'LS', 'WP', 'PRP$', 'CC ', 'DT', '``', ',', 'VB', 'UH', '-RRB-', 'WDT', "''". Meanwhile, in the test data, there are only 39 classes, where the test data does not have the 'WP$', 'LS', and 'UH' classes. The number of words in each class is visualized in the image below. From the picture shown, it can be seen that the distribution of words for each class in the train data and test data is very similar.



Distribution of train data



Distribution of test data

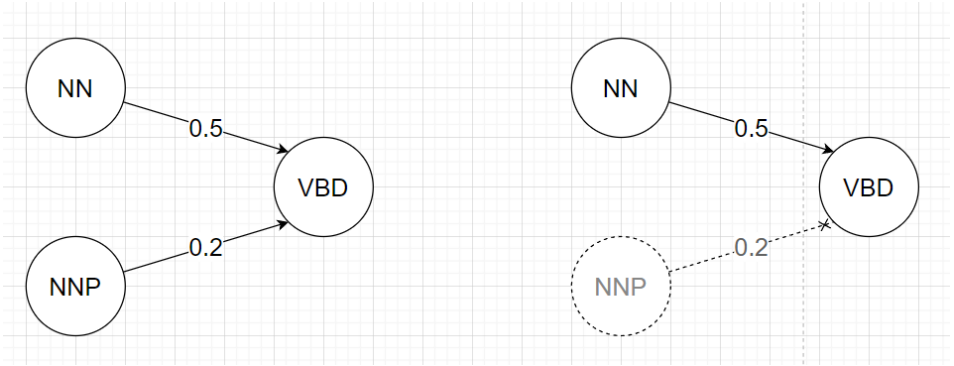
Hidden Markov Model

Hidden Markov Model (HMM) is a Hidden Markov algorithm which assumes that there are hidden states in the model. In the case of part-of-speech tagging, the nodes of the Markov model are the part-of-speech tags and the vertices that connect each node to each other are the transition probabilities. While the hidden state is the probability of each node emitting a certain word.

Viterbi Algorithm

Viterbi algorithm is commonly used in conjunction with HMM. This algorithm is a dynamic programming algorithm that works better than the greedy algorithm. The main point of the Viterbi algorithm is the elimination or selection of the path that has the lowest probability and saves the path that has the highest probability. When there are 2 nodes with different probabilities that lead to a node, only the node

with the highest probability will be saved, while the other nodes will be deleted and will not be used again in the future.



Viterbi algorithm prioritizes the highest probability going to a node

Train Model

To train the HMM model, the *fit()* function is called from the HMM class. the *fit()* function accepts training data in pandas dataframe format as shown below. From the picture below, it can also be seen that the training data has a sequence of *{word}_{part-of-speech}* format.

index	text
0	Natural_JJ language_NN processing_NN -LRB-_-LRB- NLP_NN -RRB-_-RRB- is_VBZ a_DT field_NN of_IN computer_NN science_NN ,_, artificial_JJ intelligence_NN -LRB-_-LRB- also_RB called_VBN machine_NN learning_NN -RRB-_-RRB- ,_, and_CC linguistics_NNS concerned_VBN with_IN the_DT interactions_NNS between_IN computers_NNS and_CC human_JJ -LRB-_-LRB- natural_JJ -RRB-_-RRB- languages_NNS ._.
1	Specifically_RB ,_, it_PRP is_VBZ the_DT process_NN of_IN a_DT computer_NN extracting_VBG meaningful_JJ information_NN from_IN natural_JJ language_NN input_NN and\/or_CC producing_VBG natural_JJ language_NN output_NN ._.
2	In_IN theory_NN ,_, natural_JJ language_NN processing_NN is_VBZ a_DT very_RB attractive_JJ method_NN of_IN human_JJ --_: computer_NN interaction_NN ._.
3	Natural_JJ language_NN understanding_NN is_VBZ sometimes_RB referred_VBN to_TO as_IN an_DT AI-complete_JJ problem_NN because_IN it_PRP seems_VBZ to_TO require_VB extensive_JJ knowledge_NN about_IN the_DT outside_JJ world_NN and_CC the_DT ability_NN to_TO manipulate_VB it_PRP ._.
4	Whether_NNP NLP_NNP is_VBZ distinct_JJ from_IN ,_, or_CC identical_JJ to_TO ,_, the_DT field_NN of_IN computational_JJ linguistics_NNS is_VBZ a_DT matter_NN of_IN perspective_NN ._.
5	The_DT Association_NNP for_IN Computational_NNP Linguistics_NNPS defines_VBZ the_DT latter_JJ as_IN focusing_VBG on_IN the_DT theoretical_JJ aspects_NNS of_IN NLP_NNP ._.
6	On_IN the_DT other_JJ hand_NN ,_, the_DT open-access_JJ journal_NN ``_`` Computational_JJ Linguistics_NNS "_" ,_, styles_NNS itself_PRP as_IN ``_`` the_DT longest_JJS running_VBG publication_NN devoted_VBD exclusively_RB to_TO the_DT design_NN and_CC analysis_NN of_IN natural_JJ language_NN processing_NN systems_NNS "_" -LRB-_-LRB- Computational_NNP Linguistics_NNP -LRB-_-LRB- Journal_NNP -RRB-_-RRB- -RRB-_-RRB- Modern_NNP NLP_NNP algorithms_NNS are_VBP grounded_VBN in_IN machine_NN learning_NN ,_, especially_RB statistical_JJ machine_NN learning_NN ._.

Train data

Several functions are executed by the model in the training process. In order, the functions called by their functions are as follows:

```
def fit(self, data):
  self._data = data.copy().swifter.apply(self.startEndMarker)
  self._sentencePos = [self.tokenize(sentence) for sentence in self._data["text"].values]
  self._sentenceSentencePosPos = [self.tokenizeAndTranspose(sentence) for sentence in self._data["text"].values]

  self.countPosTagAndVocab()
  self.initiateMatrix()
  self.smoothMatrix()
```

Training code snippet

a. startEndMarker(): this function is useful for adding a new *{word}_{part-of-speech}* token at the beginning and end of a sentence. *<S>_<S>* is added at the beginning of the sentence and *</S>_</S>* is added at the end of the sentence. The use of this function aims to show the probability that

a part-of-speech tag is at the beginning of a sentence or at the end of a sentence.

```
def startEndMarker(self, sentence):
  return str("<s>_<S> ") + sentence + str(" </s>_</S>")
```

b. tokenize(): the *tokenize()* function is useful for tokenizing sentences into a list containing *{word}_{part-of-speech}* tokens. Later, the list generated from this function will be used to initialize the transition probability and emission probability matrix.

```
def tokenize(self, sentence):
  tokenized = [token.split("_") for token in sentence.split()]
  return tokenized
```

c. tokenizeAndTranspose(): the function works similarly to the *tokenize()* function, but the difference is that this function separates word and part-of-speech. Later, this function will return 2 lists, namely a list of word tokens and a list of token part-of-speech of a sentence.

```
def tokenizeAndTranspose(self, sentence):
  tokenized = [token.split("_") for token in sentence.split()]
  transposed = [list(i) for i in zip(*tokenized)]
  return transposed
```

d. countPosTagAndVocab(): this function aims to generate 2 lists; the first list contains every part-of-speech tag in the dataset and the second list contains every word in the dataset.

```
def countPosTagAndVocab(self):
  temp = []
  for i in range(len(self._sentenceSentencePosPos)):
    temp += self._sentenceSentencePosPos[i][1]

  posList = dict((x, temp.count(x)) for x in set(temp))
  self._posList = list(posList.keys())

  temp = []
  for i in range(len(self._sentenceSentencePosPos)):
    temp += self._sentenceSentencePosPos[i][0]

  vocabList = dict((x.lower(), temp.count(x.lower())) for x in set(temp))
  self._vocabList = [vocab.lower() for vocab in list(vocabList.keys())]
```

e. initiateMatrix(): is a function whose job is to initialize the transition probability matrix and emission probability matrix. This function generates a probability matrix with pandas dataframe format, where the rows and columns are arranged according to the list generated by the *countPosTagAndVocab()* function.

```
def initiateMatrix(self):
  print("Initiating transition probability")
  self._transitionProb = pd.DataFrame(
      data = np.zeros((len(self._posList)-1, len(self._posList)-1), dtype = float),
      index = list(set(self._posList) - set(['</S>'])),
      columns = list(set(self._posList) - set(['<S>']))
    )

  for i in range(len(self._sentenceSentencePosPos)):
    for j in range(len(self._sentenceSentencePosPos[i][1])):
      if j != 0:
        context = self._sentenceSentencePosPos[i][1][j]
        given = self._sentenceSentencePosPos[i][1][j-1]
        self._transitionProb[context].loc[given] += 1

  self._transitionProb = self._transitionProb.div(self._transitionProb.sum(axis=1), axis=0)
  print("Done!")

  print("Initiating emission probability")
  self._emissionProb = pd.DataFrame(
      data = np.zeros((len(self._posList), len(self._vocabList)), dtype = float),
      index = self._posList,
      columns = self._vocabList
  )

  for i in range(len(self._sentencePos)):
    for j in range(len(self._sentencePos[i])):
      vocab = self._sentencePos[i][j][0]
      pos = self._sentencePos[i][j][1]
      self._emissionProb[vocab.lower()].loc[pos] += 1

  self._emissionProb = self._emissionProb.div(self._emissionProb.sum(axis=1), axis=0)
  print("Done!")
```

f. smoothMatrix(): the *smoothMatrix()* function is a function that I created myself, considering that there is a possibility that a part-of-speech tag has a transition probability of 0. A transition probability of 0 can cause deadlocks in my code, so the transition probability must be changed. The value I use is 0.0001607717041800643 divided by 2, where 0.0001607717041800643 is the smallest transition probability value in the transition probability matrix. This value is chosen in the hope that it does not interfere with calculations when running the Viterbi algorithm.

```
def smoothMatrix(self):
  self._transitionProb = self._transitionProb.replace(0, 0.0001607717041800643/2)
```

Test Model

Broadly speaking, the *predict()* function is divided into 2 blocks. The first block is tasked with improving the emission probability matrix. This is important considering that some of the words in the previous test data did not appear in the train data. The second block handles the HMM process with the Viterbi algorithm. But I coded the HMM with the Viterbi algorithm in such a way that I didn't need a backtracking process.

```
def predict(self, data, dataFrame = True, printStep = False, getResult = False):
  print("preprocessing data")
  self._predictionList = []

  if dataFrame:
    self._predictData = data.copy().swifter.apply(self.startEndMarker)
    self._sentenceSentencePosPosPredict = [self.tokenizeAndTranspose(sentence) for sentence in self._predictData["text"].values]
  else:
    rebuildSentence = ""
    preprocessedData = "<s> " + data + " </s>"
    preprocessedData = preprocessedData.split()
    preprocessedData = [pdat + "_MASK" for pdat in preprocessedData]

    for i in preprocessedData:
      rebuildSentence = rebuildSentence + i + " "

    rebuildSentence.strip()
    self._sentenceSentencePosPosPredict = [self.tokenizeAndTranspose(sentence) for sentence in [rebuildSentence]]

  self._sentencePredict = [sentence[0] for sentence in self._sentenceSentencePosPosPredict]
  self._sentenceTag = [sentence[1] for sentence in self._sentenceSentencePosPosPredict]



  for i in range(len(self._sentencePredict)):
    for j in range(len(self._sentencePredict[i])):
      if self._sentencePredict[i][j].lower() not in self._emissionProb.columns:
        self._emissionProb[self._sentencePredict[i][j].lower()] = 1
        self._emissionProb[self._sentencePredict[i][j].lower()]['<S>'] = 0
        self._emissionProb[self._sentencePredict[i][j].lower()]['</S>'] = 0
  print("Done!")
```

First block of the predict() function

```
  print("Processing HMM")
  for i in range(len(self._sentencePredict)):
    listPath = []
    listProb = []

    for j in range(len(self._sentencePredict[i])):
      tempProb = []
      tempPath = []

      if j == 0:
        listPath.append(['<S>'])
        listProb.append(1)
      else:
        prevKeyword = self._sentencePredict[i][j-1].lower()
        currentKeyword = self._sentencePredict[i][j].lower()
        prevTag = [prev[-1] for prev in listPath]
        currentTag = self._emissionProb[currentKeyword][self._emissionProb[currentKeyword] > 0].index.to_list()

        for k in range(len(currentTag)):
          bestProb = 0
          bestTag = None

          for l in range(len(prevTag)):
            if self.countProb(prevTag[l], currentTag[k], currentKeyword) * listProb[l] > bestProb:
              bestProb = self.countProb(prevTag[l], currentTag[k], currentKeyword) * listProb[l]
              bestTag = [prevTag[l], currentTag[k]]
            if l == len(prevTag)-1:
              for path in listPath:
                if path[-1] == prevTag[l]:
                  tempPath.append(path[:-1] + bestTag)
                  tempProb.append(bestProb)

        listPath = tempPath
        listProb = tempProb
        if printStep is True: print(tempPath, tempProb)

    self._predictionList.append(listPath[0])

  print("Done!")
  if dataFrame: print("Accuracy: ", self.accuracy())
  if getResult is True: return self.getPrediction(dataFrame)
```

Second block of the predict() function

Results

The HMM and Viterbi algorithms are actually only used when predicting data. Therefore the code snippet in this data prediction section is quite complicated. In my code, predicting data can be executed by calling the *predict()* function, which takes 4 inputs namely:

1. data: can be a dataframe or a single sentence.
2. dataFrame (default = True): must be changed to False when the data is in the form of a single sentence. Returns accuracy when value is True.
3. printStep (default = False) : determines whether to print Viterbi algorithm steps.
4. getResult (default = False): returns a dataframe containing the predicted result.

The following is an example of using the *predict()* function and predicting the results of testing data, reaching an **accuracy score of 90.62%**. Also attached is the use of the *predict()* function when the input is a single sentence and when returning the steps of the Viterbi algorithm.

```
res = hmm.predict(dfTest, dataFrame = True, printStep = False, getResult = True)
display(res)
```

```
preprocessing data
Done!
Processing HMM
Done!
Accuracy:  0.9062020600482139
```

1 to 25 of 171 entries Filter

index	text
0	In_IN computational_NNP linguistics_NNS ,_, word-sense_JJ disambiguation_NN -LRB-_-LRB- WSD_NN -RRB-_-RRB- is_VBZ an_DT open_JJ problem_NN of_IN natural_JJ language_NN processing_NN ,_, which_WDT governs_VBZ the_DT process_NN of_IN identifying_VBG which_WDT sense_NN of_IN a_DT word_NN -LRB-_-LRB- i.e._FW meaning_NN -RRB-_-RRB- is_VBZ used_VBN in_IN a_DT sentence_NN ,_, when_WRB the_DT word_NN has_VBZ multiple_JJ meanings_NNS -LRB-_-LRB- polysemy_NN -RRB-_-RRB- ._.
1	The_DT solution_NN to_TO this_DT problem_NN impacts_IN other_JJ computer-related_NNS writing_NN ,_, such_JJ as_IN discourse_NN ,_, improving_VBG relevance_NN of_IN search_NN engines_NNS ,_, anaphora_NN resolution_NN ,_, coherence_NN ,_, inference_NN et_FW cetera_NN ._.
2	Research_NN has_VBZ progressed_VBD steadily_RB to_TO the_DT point_NN where_WRB WSD_JJ systems_NNS achieve_VB sufficiently_RB high_JJ levels_NNS of_IN accuracy_NN on_IN a_DT variety_NN of_IN word_NN types_NNS and_CC ambiguities_NNS ._.
3	A_DT rich_JJ variety_NN of_IN techniques_NNS have_VBP been_VBN researched_VBN ,_, from_IN dictionary-based_JJ methods_NNS that_IN use_VBP the_DT knowledge_NN encoded_NN in_IN lexical_JJ resources_NNS ,_, to_TO supervised_JJ machine_NN learning_NN methods_NNS in_IN which_WDT a_DT classifier_NN is_VBZ trained_VBN for_IN each_DT distinct_JJ word_NN on_IN a_DT corpus_NN of_IN manually_RB sense-annotated_JJ examples_NNS ,_, to_TO completely_RB unsupervised_JJ methods_NNS that_IN cluster_NN occurrences_NN of_IN words_NNS ,_, thereby_RB inducing_JJ word_NN senses_NNS ._.

Dataframe input with accuracy score and predicted part-of-speech tags in dataframe format

```
sentence = "Natural Language Processing Laboratory in Nara Institute of Science and Technology"
res = hmm.predict(sentence, dataFrame = False, printStep = True, getResult = True)
display(res)
```

```
preprocessing data
Done!
Processing HMM
[['<S>', 'NNP'], ['<S>', 'JJ']] [0.00020810327292744246, 0.00293954353683873]
[['<S>', 'NNP', 'NNP'], ['<S>', 'JJ', 'NN']] [3.3719794726192106e-07, 3.063187440033033e-05]
[['<S>', 'JJ', 'NN', 'NNP'], ['<S>', 'JJ', 'NN', 'NN'], ['<S>', 'JJ', 'NN', 'VBG']] [3.6559195215154354e-10, 4.243905407647011e-08, 1.8857118931689873e-09]
[['<S>', 'JJ', 'NN', 'NN', 'JJS'], ['<S>', 'JJ', 'NN', 'NN', ','], ['<S>', 'JJ', 'NN', 'NN', 'RB'], ['<S>', 'JJ', 'NN', 'NN', '-RRB-'], ['<S>', 'JJ', 'NN', 'NN', 'FW'], ['<S>', 'JJ', '
[['<S>', 'JJ', 'NN', 'VBG', '-LRB-', 'FW'], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN']] [1.3971569359056677e-12, 2.4595033912134216e-10]
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'JJS'], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', ','], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'RB'], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', '-RRB-
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP']] [2.857756271108408e-15]
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN']] [4.091302329812631e-17]
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NNP'], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NN']] [1.4946676319776717e-21, 1.3468289030575007e-20]
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NN', 'CC']] [3.8584195790302694e-22]
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NN', 'CC', 'NNP'], ['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NN', 'CC', 'NN']] [5.953419879124474e-26, 2.
[['<S>', 'JJ', 'NN', 'VBG', 'NN', 'IN', 'NNP', 'NNP', 'IN', 'NN', 'CC', 'NN', '</S>']] [2.1027432000444447e-29]
Done!
'Natural_JJ Language_NN Processing_VBG Laboratory_NN in_IN Nara_NNP Institute_NNP of_IN Science_NN and_CC Technology_NN'
```

Single sentence input, returning the Viterbi algorithm's steps and the final part-of-speech tag result