CS6320: Assignment 2

https://github.com/cs6320-501-fall22/assignment2-group18
Group 18 Kevin Tak Hay Cheung (kxc220019) Humza Salman (mhs180007)

Introduction

The project aims to implement a Hidden Markov Model to perform part-of-speech tagging. We will build multiple n-gram models and tune the corresponding smoothing parameters for the Viterbi algorithm so that we can assess the best pick for our POS tag predictions. After the initial modeling, we will examine the per token accuracy rates to identify room for improvement in the algorithm. Ultimately, we will pass the test data to the modified Viterbi Algorithm to generate POS tags for the sequences. Our final model utilized a trigram Viterbi algorithm to get an accuracy of 95.062% for POS tag predictions.

The following sections will provide details of the modeling process such as data preprocessing, implementation details, and the experiment design and evaluation details. We will conclude with our results and analysis of the model.

Data

We are using data from the Penn Treebank which includes POS tagged sentences to train and validate our models. It consists of over 1 million words - each with a labeled POS tag. The data is split into an approximately 60/20/20 split for the train/dev/test sets. A figure is provided to give an accurate count of the data present.

	Count
Train	696476
Dev	243022
Test	236583
Total	1176081

The datasets provided were already tokenized for our convenience. When processing the dataset we did an inner join to match the token to the tag to form a token-tag pair. When loading in data we appended each token-tag pair to the starting with the [‘-DOCSTART-’, ‘O’] token-tag pair until another [‘-DOCSTART-’, ‘O’] token-tag pair was encountered, where we appended a [‘-DOCEND-’, ‘END’] token-tag pair. This was done to keep track of the start of each document. We repeated this process until the data was finished loading into a list.

Implementations

To utilize the Viterbi algorithm to obtain the POS-tag for a sequence, we create a Python class named POS_Tagger. It is a generalized class that can train different n-gram models for the POS-tagging algorithm. There are several methods to help with the POS-tagging, including:

- ***__init__***: It takes the parameter *n* to indicate the n-gram model will be used in all methods
- ***train***: it takes the training dataset (the token-tag pairs) and computes the count of the POS-tag unigram and emission unigram
- ***smoothing_k***: It takes the smoothing parameters and converts the counts from train methods to smoothed transition and emission probabilities, which will be used in the Viterbi algorithm .
- ***inference***: It takes a sequence and returns the POS-tags with the highest likelihood

This section will further discuss these class methods. Most implementations are completed from scratch without relying on any third-party NLP packages. Also, we decided to go with the python dictionaries data structure due to their simplicity and familiarity.

POS_Tagger.train

The purpose of this class method is to handle the unknown word and count the n-grams that will be used in the modeling process.

Unknown Words

To account for unseen words, we decided to replace the token of the token-tag pair with the <UNK> token. By doing so, when the model encounters any unseen words during the inference, instead of returning a zero probability, the unseen word will be replaced by the <UNK> tag and the probability of this tag will be used. We went forward with our approach from the previous assignment - random word selection. This method selects words at random and replaces their occurrences with the unknown token. We utilize the random function from the python library ‘random’ and use a uniform distribution to select a percentage of words to replace. We will experiment with 0.5%, 0.75%, and 1% for the percentage selection and compare the outcome.

POS Tag Unigram Count

The POS-tag unigram probability will be used in various ways throughout the modeling process. Therefore, we must compute the count for each POS-tag in the corpus so that we can obtain the probability. We utilized python’s dictionary to store all the counts. Our algorithm will loop through each token-tag pair and check if the POS-tag exists in the dictionary. If it does not exist, we create a new key and set the value to 1. Otherwise, we will add 1 to the currently stored value. The details can be found in the following code snippet.

Emission Unigram Count

We also need the token count under each POS-tag in the corpus for the emission probabilities. We used python’s dictionary structure to store all the tags and then a nested dictionary to store the token counts under that tag. We go through each token and append the count for the token-tag pair if it exists. The details can be found in the following code snippet.

```
for sen in data:
    for word in sen:

        # POS Tag Unigram
        if word[1] not in self._pos_unigram_cnt:
            self._pos_unigram_cnt[word[1]] = 1
        else:
            self._pos_unigram_cnt[word[1]] += 1

        # Emission Unigram
        if word[1] not in self._emission_unigram_cnt:
            self._emission_unigram_cnt[word[1]] = {i : 0 for i in self._word_list}

        if word[0] not in self._emission_unigram_cnt[word[1]]:
            self._emission_unigram_cnt[word[1]][word[0]] = 1
        else:
            self._emission_unigram_cnt[word[1]][word[0]] += 1
```

Prior-Gram and N-Gram Counts

For the purposes of explanation we will consider our implementation with N = 3. First, we pad the start of the document with the appropriate number of [‘-DOCSTART-’, ‘O’] token-tag pair when computing the prior and n-gram counts. Then, we construct the prior-gram counts by joining tags that appear in a N-1 sequence. For example for N=3, ‘O PRP VBP TO VB NNS’ would be padded to ‘O O PRP VBP TO VB NNS’ such that some of the prior-grams would be: ‘O O’, ‘O PRP’, ‘PRP VBP’. We would then go through the document and compute the count for each prior-gram.

For our n-gram counts we add all prior-grams to our n-gram count dictionary as keys and then create a nested dictionary for each prior-gram. The nested dictionary contains every possible POS tag that could follow the prior-gram. Then we go through the document and increment the count for the n-grams that exist. The details can be found in the following code snippet.

```
for sen in data:
        sen = [('-DOCSTART-', 'O') for _ in  range(self._n - 2)] + sen

        for i in range(0, len(sen)- self._n + 1):

            # Create n-1 gram
            prior_ngram = " ".join([ngram[1] for ngram in sen[i:(i+self._n-1)]])
            if prior_ngram not in self._prior_ngram_cnt:
                self._prior_ngram_cnt[prior_ngram] = 1
            else:
                self._prior_ngram_cnt[prior_ngram] += 1

            # Create ngram
            if prior_ngram not in self._ngram_cnt:
                self._ngram_cnt[prior_ngram] = {j : 0 for j in self._pos_unigram_cnt}

            if sen[i+self._n-1][1] not in self._ngram_cnt[prior_ngram]:
                self._ngram_cnt[prior_ngram][sen[i+self._n-1][1]] = 1
            else:
                self._ngram_cnt[prior_ngram][sen[i+self._n-1][1]] += 1
```

POS_Tagger.smoothing_k

The purpose of this method is to handle the unseen emission (token-tag pair) and POS-tag n-gram, whose components are seen in the training set, using Add-K smoothing. It is a technique that adds k occurrences to each entry of the emission unigram count or the n-gram count and then normalizes it to create a smoothed probability distribution. This method will take two parameters, k1 and k2, to perform the smoothing for emission unigram and POS-tag n-gram respectively. The details can be found in the following code snippet .

```
def smoothing_k(self, k1, k2):
    # Transform n-gram count to probability
    self._k1 = k1
    self._k2 = k2

    self._ngram_prob = copy.deepcopy(self._ngram_cnt)
    self._emission_prob = copy.deepcopy(self._emission_unigram_cnt)
    for i in self._prior_ngram_cnt:
      for j in self._ngram_prob[i]:
        self._ngram_prob[i][j] = (self._ngram_prob[i][j] + k1) / (self._prior_ngram_cnt[i] + k1 * len(self._pos_unigram_cnt))

    for i in self._emission_prob:
      for j in self._emission_prob[i]:
        self._emission_prob[i][j] = (self._emission_prob[i][j] + k2) / (self._pos_unigram_cnt[i] + k2 * len(self._word_list))
```

POS_Tagger.inference

For the Viterbi Matrix we utilized a dictionary structure. The key of the dictionary is all the possible POS tags and the value would be a list $L_{tag.}$ The length of L_{tag} would be the length of the sequence (plus appropriate number of the padded -DOCSTART- tokens). Each element in L_{tag} (aka $L_{tag ,t}$) is initialized to [-infinity, None] - the first element represents the log-likelihood of the sequence having the tag at position t and the second element represents the prior POS tag at position t-1.

After creating the Viterbi Matrix, we need to preprocess the sequence - it requires padding with appropriate numbers of 'O' tokens to signify the document start for different n-gram choices and also ensuring we take unknown words into account with the '<UNK>' token.

To start the Viterbi Algorithm, each $L_{o, t}$ at the beginning of the sequence will be initialized with [0, 'O'] until we reach the last -DOCSTART- followed by the first actual token in the sequence. For each position t, we identify all possible POS-tags in that position by examining $L_{tag, t}$[0]. If $L_{tag, t}$[0] is non-negative-infinity, it means that there is a possible prior-gram preceding the POS-tag. Next, we will combine the prior-gram (using information from $L_{tag, t}$[1]) and the POS-tag to form an n-gram. Using the transition probability distribution of the newly formed n-gram, we can identify all possible POS-tags in the position t+1. Combining the emission probability for these possible POS-tags, we can compute the log likelihood for the sequence at position t+1 for all possible POS tags. If the log-likelihood is larger than the current likelihood for the possible tag at t+1, we will update $L_{tag, t+1}$. The details can be found in the following code snippet.

```
def inference(self, sequence):
    # Vit_Matrix
    # Dictionary Key : POS Tag
    # Each Value in Dictionary is a list - each element is the list is also a list, representing the status at position t
    # List is consisted of tuple - 1st : Prob 2nd : Prior Tag
    Vit_Matrix = {i: [ [float("-inf"), None] for _ in range(len(sequence) + max(0, self._n - 2)) ]   for i in self._pos_unigram_cnt}

    sequence_unk = ['O' for _  in  range(self._n - 2)] + [ word if (word in self._word_list and word not in self._unk_list) else "<UNK>"  for word in sequence]
    sequence = ['O' for _  in  range(self._n - 2)]  + sequence

    for t in range(max(1, self._n - 1)):
      Vit_Matrix['O'][t][0] = 0
      Vit_Matrix['O'][t][1] = 'O'

    for t in range(self._n - 2, len(sequence_unk) - 1): # Check each position in the sequence
      try:
        check_float = float(sequence[t].replace(',', '').replace(':', ''))

        for tag in self._pos_unigram_cnt:
          if tag != 'CD':
            Vit_Matrix[tag][t][0] = float("-inf")
            Vit_Matrix[tag][t][1] = None

      except:
        pass

      for tag in self._pos_unigram_cnt: #For each position in the sequence, check if there is any non-zero prob for each pos tag
        if Vit_Matrix[tag][t][0] != float("-inf"):

          prior_ngram = [tag]
          tmp_tag = Vit_Matrix[tag][t][1]

          for tag_reverse in range(t-1, t - self._n + 1, -1):
            prior_ngram = [tmp_tag] + prior_ngram
            tmp_tag = Vit_Matrix[tmp_tag][tag_reverse][1]

          prior_ngram = " ".join(prior_ngram)

          if prior_ngram in self._ngram_prob:
            search_dict =  self._ngram_prob[prior_ngram]
          else:
            search_dict = self._pos_unigram_prob

          for next_tag in search_dict:

            if next_tag in search_dict:
                Transition_Prob = search_dict[next_tag]
            else:
                Transition_Prob = self._k1 / (self._k1 * len(self._pos_unigram_cnt))

            if sequence_unk[t+1] in self._emission_prob[next_tag]:
                Emission_Prob = self._emission_prob[next_tag][sequence_unk[t+1]]
            else:
                Emission_Prob = self._k2 / (self._pos_unigram_cnt[next_tag] + self._k2 * len(self._word_list))

            Next_Prob = Vit_Matrix[tag][t][0] + math.log10(Transition_Prob)  + math.log10(Emission_Prob)

            if Next_Prob > Vit_Matrix[next_tag][t+1][0]:
                Vit_Matrix[next_tag][t+1][0] = Next_Prob
                Vit_Matrix[next_tag][t+1][1] = tag

    self._Vit_Matrix = Vit_Matrix
```

Once this process is finished we trace back the best route that will give us the most probable POS tagging sequence. We find the POS tag at the end of the sequence with the maximum likelihood using $L_{tag, t}[0]$ and backtrack using the prior-gram tag, $L_{tag, t}[1]$ - with our dictionary structure making it very convenient. The details can be found in the following code snippet.

```
    #Trace Back the best route
    cur_max = float("-inf")
    cur_tag = None
    for tag in Vit_Matrix:
        if Vit_Matrix[tag][len(sequence_unk)-1][0] > cur_max:
            cur_max = Vit_Matrix[tag][len(sequence_unk)-1][0]
            cur_tag = tag

    result = [cur_tag]
    for t in range(len(sequence_unk)-1, 0, -1):
        result = [Vit_Matrix[cur_tag][t][1]] + result
        cur_tag = Vit_Matrix[cur_tag][t][1]

    final_result = list(result[self._n - 2 :-1])
    final_seq = list(sequence_unk[self._n - 2 :-1])
```

Experimental Design and Evaluations

Our motivation for the experiment came from wanting to assign POS tags to documents so that we are able to distinguish the part-of-speech of a word. POS tags give insight into how a word is being used within a phrase, sentence, or document and they are used for insight into corpus searches, text analysis tools and algorithms.

Our goal for this experiment was to implement a Hidden Markov Model using the Vanilla Viterbi algorithm to achieve accuracies greater than 94%. We decided to experiment with N=3…8 for the n-grams in our HMMs as the neighboring words of a sequence can impact the POS-tag assigned to a word. We expected that the higher n-gram counts would require more computational power, but also yield greater accuracies until a certain threshold was reached. We expected that certain types of words would be misclassified with their POS tags so we planned to utilize an evaluation script to help aid the development of our model.

Our evaluation script uses our HMM models to compute the predicted sequences using the POS_Tagger.inference method described within this report. We also load in the actual POS-tagging sequence and join the predicted and actual POS tags within a Pandas DataFrame. We use numpy to compute the Per-Tag accuracy so that we can evaluate our model on a per-tag basis as we believed this was the optimal way to improve our model.

Results

In this section, we will present the initial result for several n-gram base models (varying N and keeping all other parameters constant). Next, we will pick the optimal base model that we would like to move forward with for hyperparameter tuning. In the end, we will present our best model and identify room for improvement in the next section.

Base Model

By setting all other parameters the same, we compared the performance of the n-gram models directly. We divided the training set into 90/10 subsets, ran the Vanilla Viterbi Algorithm and recorded the accuracy rate for each model. The following chart summarizes the performance of these base models.

Ngram	UNK	k1	k2	Average Accuracy
2	0.01	0.01	0.01	94.52%
3	0.01	0.01	0.01	94.64%
4	0.01	0.01	0.01	93.68%
5	0.01	0.01	0.01	91.92%
6	0.01	0.01	0.01	89.55%
7	0.01	0.01	0.01	89.51%
8	0.01	0.01	0.01	89.78%

From the chart, we observed that the trigram model performed the best in the initial modeling. The next step is to tune the hyper parameters for the trigram model. The hyperparameter requires tuning includes the smoothing parameters, k1 and k2 and the unknown word selection percentage, unk.

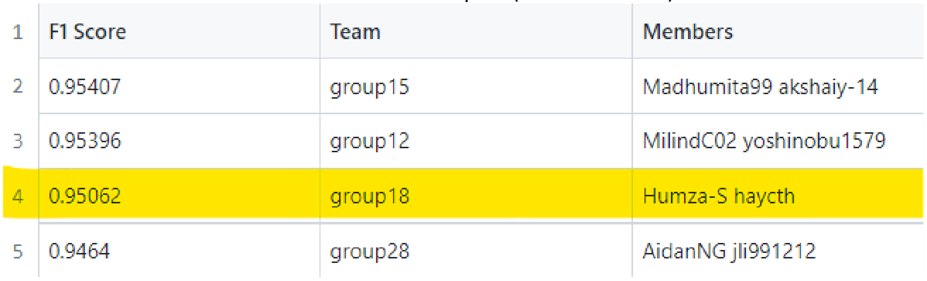
We have experimented with different setups of these parameters and the results of the best performing models are attached below. These models are trained on the entire training sets and tested on the development sets.

Ngram	UNK	k1	k2	Accuracy
3	0.01	0.1	0.001	94.19%
3	0.01	0.01	0.001	94.17%
3	0.0075	0.1	0.001	94.13%
3	0.005	0.1	0.001	94.12%
3	0.0075	0.01	0.001	94.11%
3	0.005	0.01	0.001	94.10%
3	0.01	0.001	0.001	94.09%
3	0.01	0.1	0.01	94.09%
3	0.01	0.01	0.01	94.07%
3	0.005	0.1	0.01	94.03%

From the chart, we identified the best model is the one with k1 = 0.1, k2 = 0.001 and unknown word selection percentage = 0.01. On top of tuning the optimal parameters, we also modified the Viterbi Algorithm a bit (the details can be found in the Error analysis section), and our finalized model's performance trained by the train set has an accuracy rate of 94.92% in the dev set.

After training the model, we used the model to predict the POS-tag for the test set. We uploaded the test_y.csv to github and resulted in an accuracy rate of 95.062%.

Team-Name: Group 18 (Kevin & Humza)



	F1 Score	Team	Members
2	0.95407	group15	Madhumita99 akshaiy-14
3	0.95396	group12	MilindC02 yoshinobu1579
4	0.95062	group18	Humza-S haycth
5	0.9464	group28	AidanNG jli991212

Screenshot Taken 10/17/2022 12:53pm

Analysis

Error Analysis

With the optimal model, we will perform an error analysis by examining the per-tag accuracies from the models. The following chart shows the accuracy rate for each POS-tag.

POS-tag	Number of tags	Trigram Accuracy	POS-tag	Number of tags	Trigram Accuracy	POS-tag	Number of tags	Trigram Accuracy	POS-tag	Number of tags	Trigram Accuracy
,	12558	100%	EX	210	96%	O	462	100%	VB	6790	94%
:	1187	100%	FW	22	23%	PDT	46	96%	VBD	7907	93%
.	10078	100%	IN	25237	97%	POS	2069	99%	VBG	3675	86%
''	1664	100%	JJ	15277	92%	PRP	4333	99%	VBN	5220	82%
(	309	100%	JJR	837	89%	PRP$	2162	100%	VBP	3142	90%
)	316	100%	JJS	511	89%	RB	7949	88%	VBZ	5296	95%
#	46	100%	LS	10	80%	RBR	463	67%	WDT	1080	96%
``	1711	100%	MD	2494	100%	RBS	125	81%	WP	614	99%
$	2051	99%	NN	34400	94%	RP	662	79%	WP$	39	100%
CC	6157	99%	NNP	23113	92%	SYM	25	76%	WRB	534	100%
CD	9509	94%	NNPS	498	60%	TO	5791	100%			
DT	21003	99%	NNS	15415	94%	UH	24	25%			

After comparing the actual tags and predictions, we identified several ways to further improve the model. For example, we noticed that a significant portion of CD tags are misclassified. The following chart includes several examples.

Word	Unseen?	Actual POS-tag	Predicted POS-tag
881,969	Yes	CD	JJ
21.6	Yes	CD	NN
Four	No	CD	NNP
354,600	Yes	CD	VBG
109,000	Yes	CD	JJ
2.9495	Yes	CD	JJ
2.9429	Yes	CD	JJ
142.43	Yes	CD	JJ

It seems that a huge portion of these misclassified tags are unseen in the training corpus. It makes sense since there are infinite possible numbers. These unknown numbers will be replaced with the <UNK> tag during the inference process and be assigned with a probability based on the unknown tag distribution across different POS-tags. On top of that, the subsequent probability distribution after the positions of the unseen numbers will be biased since the CD probability distribution is not applied correctly.

When we examined the training data, we realized all numbers are attached with CD. As a result, we proposed two methods to resolve the misclassification.

1. After the inference, we manually update the predicted POS-tags of the numbers to 'CD'.
2. During the inference, when we compute the transition and emission probabilities of a number, we prune the paths of all other POS-tags except 'CD'.

The first method would effectively correct the predictions of all numbers. However, since the POS-tags are still misclassified during the inference, the subsequent likelihood after the number token's position will still be biased. For the second method, since the POS-tags are corrected during the inference, the subsequent probability will be based on the correct n-gram with 'CD' tag. Therefore, we decided to go with the second method. With this adjustment, we were able to increase the CD accuracy rate from 94% to 99%.

Important Features

Although our model mostly depends on the transitional and emission distributions without relying on additional features, we did identify some useful features that should be implemented if we decided to go with the MEMM model. An example would be checking if the token belongs to any closed class lexion, such as '$', "WRB'.

Additionally, some contextual clues would be useful. In our error analysis, we noticed that prefixes and suffixes are very impactful for POS-tagging. A suffix of '-ful' or '-able' is very likely to be an adjective. However, we cannot solely depend on these clues because they sometimes can be misleading. A good example would be the token 'able', which should be classified as a verb but it would be an adjective under the suffix rule.

Lastly, the contextual clues, such as the nearby predicted POS-tag or observed tokens, can be helpful too. Another observation from the error analysis is that the model always misclassified the adjective (JJ) class and the past participle (VBP) class to each other. It is because the past participle form of a verb is always used as an adjective. A solution is to include the neighboring features such as whether we can find any 'VBP' or 'VBZ', 'NN' in the prior positions since it is more likely to find a 'VBZ' or 'VBP' close to the past participle and more likely to find a 'NN' after adjective.

Conclusion

Ultimately, our team decided to move forward with the modded trigram Viterbi model with additional processing being dedicated to fix the cardinal numbers POS tag. Our findings demonstrated that we have room for improvement in our models by reclassifying certain types of tags according to the niche rules that suit them. We could also move forward with implementing an MEMM model as well. In conclusion, we arrived at an accuracy of 95.062% on the leaderboard with our final model and saw significant improvement in our accuracies as we analyzed our findings and implemented changes.

Programming Library Usage

- ***copy*** is used to create deep copies of the dictionaries storing the language model
- ***math*** is used the compute the perplexity
- ***random*** is used to perform the unknown word selection
- ***numpy*** & ***panda*** are used for general data processing

Contributions of Each Member

Kevin: Data Preprocessing (40%), POS_Tagger.train (60%), POS_Tagger.inference (60%), POS_Tagger.smoothing_k (40%), Evaluation (50%), Report (40%)

Humza: Data Preprocessing (60%), POS_Tagger.train (40%), POS_Tagger.inference (40%), POS_Tagger.smoothing_k (60%), Evaluation (50%), Report (60%)

Feedback of Project

- First and foremost we want to address the report template. We believe the report template was ambiguous and constraining in its requirements. With the 'Implementation' section of the report it mentions including important code snippets, but we felt there was not enough space to include all of the important implementation code pieces along with explanation. We had to limit our code snippets to fit within the 6 page limit which felt constraining.
- The 'Design' aspect of the report was confusing as well. It seems like a high level summary of the implementation and a description of the method we used for error analysis, which may seem redundant.

- 'Leaderboard Score' should not be included in the report - it is fun to include it but this forces us to turn in the report after 8pm Monday since it is required to be up to date. Also, it seems redundant to upload if the screenshot should match what the leaderboard shows.
- The report templates between Gradescope and elearning are not consistent, which was a bit confusing.
- We believe utilizing a github classroom environment along with the leaderboard was a fun piece to add to the assignment. Seeing other student's results promotes a friendly competition.
- We like how there was starter-code or a repo we could pull from and implement code for.
- Overall, we thoroughly enjoyed learning more about HMMs and Viterbi models during our time developing this project.