Hidden Markov Models for Part of Speech Tagging

CS 671 Final Project: Emre Yurtbay

December 8, 2021

Introduction

Consider the following sentence:

The woman ate the orange.

Any sufficiently bright fifth grader could tell you the part of speech of each word in the above sentence: ”The” is a determiner, ”woman” is a noun, ”ate” is a verb, ”the” is a determiner, and ”orange” is a noun. The ”tag sequence” for this sentence is the following:

D N V D N

At first, it would seem simple enough to train a model to build a tag sequence for any given sentence. However, consider another sentence:

I saw the orange cat.

In the first sentence, the word ”orange” functions as a noun, but in the second sentence, ”orange” is an adjective. It is easy to see that the part of speech of many words is not constant; it depends on the context in which it appears in a sentence. In part of speech (POS) tagging, our goal is to build a model where the input is a sequence of words (that is, a sentence), such as the ones above, and the output is a sequence of tags representing the parts of speech of each of the words of the input sequence. We can formalize the POS tagging problem the following way. Let the inputs be a sequence of words $x_1, x_2, \ldots, x_n$, and a ”tag set” Q, which contains all of the possible parts of speech for the language. Our output is the sequence of part of speech tags $y_1, y_2, \ldots, y_n$, where y_i corresponds with x_i, and each $y_i \in Q$. What makes this particular problem challenging to solve is something we have already alluded to: the problem of ambiguity. Many words in the the English language, and many other languages, can take different parts of speech. Part of speech tagging can therefore be thought of as a disambiguation task; a model that accurately predicts the parts of speech for each word in a sentence must effectively take into account each word’s context in a sentence. In this problem, we will use Hidden Markov Models (HMMs) to solve the POS tagging problem, and demonstrate how they are able to achieve extremely high accuracies.

I first learned about this problem while learning about stochastic processes. Hidden Markov models are an extension of the more basic Markov chain models that are some of the core structures found in stochastic processes. Part of Speech Tagging is a task that is natural for Hidden Markov models because we have a set of latent states of interest as well as observable states. The prediction task of the part of speech tagging problem is akin to the multi-class classification tasks we have become quite familiar with in CS 671. This project is also an extension on some of the latent variable models we learned in this class in the context of the EM algorithm. Finally, I also got the opportunity to learn about a new algorithm, the Viterbi algorithm, which we did not use in class but comes up all the time when statistical methods are applied to sequence data.

Related Work

I will be implementing as HMM class from scratch myself in Python 3. An implementation for HMMs is available in the natural language processing python package NLTK, but the HMMs I present here show no significant difference in performance compared to the those found in NLTK.

The Model

Hidden Markov Models (HMMs) belong to a class of models called probabilistic sequence models: given a sequence of words, they compute a probability distribution over possible sequences of labels and choose the best label. HMMs require that there exists an observable ”process” X whose outcomes are influenced by the outcomes of some unobservable ”process” Y. In our example, the observable process may be the sequence of words that make up the sentence, while the unobserved process Y is the sequence of parts of speech. We do not see the parts of speech directly; they are the latent or ”hidden” states that we are ultimately trying to recover. The HMMs we will use here consist of several components, each of which will be discussed below.

HMM Components

The first component is $Q = \{q_1, q_2, \ldots, q_n\}$, a set of n states. These states correspond with the different parts of speech, (the hidden states) that each word in a sentence can be tagged as. In grade school in America, many of us are taught that there are 9 basic parts of speech in the English language: noun, verb, article, adjective, preposition, pronoun, adverb, conjunction, and interjection. However, English is a much richer language than this, and we can define more granular parts of speech as well, such a singular proper nouns, predeterminers, gerund verbs, modals, and many, many more. *In toto*, we will be using a set Q with 36 possible ”states” when building our model.

Tag	Description	Example	Tag	Description	Example	Tag	Description	Example
CC	coord. conj.	*and, but, or*	NNP	proper noun, sing.	*IBM*	TO	“to”	*to*
CD	cardinal number	*one, two*	NNPS	proper noun, plu.	*Carolinas*	UH	interjection	*ah, oops*
DT	determiner	*a, the*	NNS	noun, plural	*llamas*	VB	verb base	*eat*
EX	existential ‘there’	*there*	PDT	predeterminer	*all, both*	VBD	verb past tense	*ate*
FW	foreign word	*mea culpa*	POS	possessive ending	*’s*	VBG	verb gerund	*eating*
IN	preposition/ subordin-conj	*of, in, by*	PRP	personal pronoun	*I, you, he*	VBN	verb past participle	*eaten*
JJ	adjective	*yellow*	PRP$	possess. pronoun	*your, one’s*	VBP	verb non-3sg-pr	*eat*
JJR	comparative adj	*bigger*	RB	adverb	*quickly*	VBZ	verb 3sg pres	*eats*
JJS	superlative adj	*wildest*	RBR	comparative adv	*faster*	WDT	wh-determ.	*which, that*
LS	list item marker	*1, 2, One*	RBS	superlatv. adv	*fastest*	WP	wh-pronoun	*what, who*
MD	modal	*can, should*	RP	particle	*up, off*	WP$	wh-possess.	*whose*
NN	sing or mass noun	*llama*	SYM	symbol	*+,%, &*	WRB	wh-adverb	*how, where*

Figure 1: A table of English Parts of Speech

Next, we have an $n \times n$ transition probability matrix A, which contains the probabilities from moving from one state (that is, part of speech) to another. For example, if adverbs (”RB”) are followed by past tense verbs (”VBD”) 60 percent of the time, $A_{RB,VBD} = 0.6$. These transitions are computed directly from the labeled training data.

The emission probability matrix E gives us the probability an observation w is generated from state q. In our case, the observations are words and the states are parts of speech. If 23 percent of all prepositions

(”IN”) in English are the word ”of”, $E_{IN,of} = 0.23$. The probabilities contained in this matrix may be slightly unintuitive, as we are calculating $P(w_i \mid t_i)$. This answers the question ”if we were going to generate a part of speech t_i such as a modal, how likely is it that the modal will be word w_i?” The MLE of the emission probability is given by

$$P(w_i \mid t_i) = \frac{C(t_i, w_i)}{C(t_i)}$$

The E matrix in our example has 36 rows. The number of columns is equal to the number of unique words in the training corpus, which can be absolutely massive; naturally, this matrix is very sparse. It is important to note that the rows of E must sum to 1.

Finally, we have an initial probability distribution π over all states. This gives us the probability of each part of speech beginning a sentence. π is a vector of 36 entries, and the sum of all entries equals 1. Some π_j can equal 0: in English, sentences starting with past tense verbs (”VBD”) are considered ungrammatical, so theoretically, $\pi_{VBD} = 0$.

The ”Order” of an HMM

One other important consideration is the ”order” of the HMM. Recall that a model that accurately predicts the parts of speech for each word in a sentence must effectively take into account each word’s context in a sentence. To predict the part of speech of a word, the model will look at a number of words before it to help determine parts of speech. If the order of the HMM is 2 and the model is trying to tag word i, it will use the tag of word $i-1$ to help tag word i. (w_i, w_{i-1}) form what is known as a bigram, and thus order 2 HMMs are also known as ”bigram HMMs”. This is also where the ”Markov property” of HMMs comes into play: in bigram HMMs, the tag for w_i only depends on the tag for w_{i-1}, and no other words before or after it. If we want to consider even more context, we can use trigram HMMs, in which tags for w_i depend not only on w_{i-1} but also w_{i-2}. Increasing the order of the HMM may lead to more accurate tags, but also to a much greater computational burden while training. HMMs can achieve remarkably accurate results even when their order is just 2 or 3.

Decoding with the Viterbi Algorithm

For a model with latent states, the task of determining the hidden variable sequence corresponding to the sequence of observations is called ”decoding.” To formalize this somewhat, given as an input an HMM and a sequence of observations $o_1, o_2, \dots o_T$ (i.e. a sentence), we want to find the most probable sequence of states $q_1, q_2, \dots q_T$. In the part of speech tagging example, the goal is to choose the tag sequence $t_1 \dots t_n$ that is most probable given the observation sequence of n words, $w_1 \dots w_n$. That is,

$$\{t^*\}_1^n = \text{arg} \max_{t_1 \dots t_n} P(t_1 \dots t_n \mid w_1 \dots w_n)$$

We can compute this probability using Bayes’ rule

$$\{t^*\}_1^n = \text{arg} \max_{t_1 \dots t_n} \frac{P(w_1 \dots w_n \mid t_1 \dots t_n) P(t_1 \dots t_n)}{P(w_1 \dots w_n)}$$

Using a standard Bayesian trick, we can actually disregard the denominator when doing this computation.

$$\{t^*\}_1^n = \text{arg} \max_{t_1 \dots t_n} P(w_1 \dots w_n \mid t_1 \dots t_n) P(t_1 \dots t_n)$$

Still this calculation is pretty difficult. We need to make a few simplifying assumptions. The first is the independence assumption:

$$P(w_1 \dots w_n \mid t_1 \dots t_n) = \prod_{i=1}^{n} P(w_i \mid t_i)$$

Algorithm 1: Trigram Viterbi.

Input: A second-order HMM M with states $Q = \{1, 2, \ldots, K\}$ given by its transition matrix A, emission probability matrix E (alphabet Σ), and probability distribution π on the (initial) states; a sequence X of length L (indexed from 0 to $L-1$).

Output: A sequence Z, with $|Z| = |X|$, that maximizes $p(Z, X)$.

$v[\ell', \ell, 1] \leftarrow (\pi_{\ell',\ell} \cdot E_{\ell'}(X_0) \cdot E_{\ell}(X_1))$ for every $\ell', \ell \in Q$;
for $i \leftarrow 2$ **to** $L-1$ **do**
  foreach $\ell' \in Q$ **do**
    foreach $\ell \in Q$ **do**
      $v[\ell', \ell, i] \leftarrow E_{\ell}(X_i) \cdot \max_{\ell'' \in Q}(v[\ell'', \ell', i-1] \cdot A_{\ell'',\ell',\ell})$;
      $bp[\ell', \ell, i] \leftarrow \text{argmax}_{\ell'' \in Q}(v[\ell'', \ell', i-1] \cdot A_{\ell'',\ell',\ell})$;
$Z_{L-2} \leftarrow \text{argmax}_{\ell' \in Q}(v[\ell', \ell, L-1])$;
$Z_{L-1} \leftarrow \text{argmax}_{\ell \in Q}(v[\ell', \ell, L-1])$;
for $i \leftarrow L-3$ **downto** 0 **do**
  $Z_i \leftarrow bp[Z_{i+1}, Z_{i+2}, i+2]$;
return Z

Finally, we need to use the order of the HMM to make a further simplifying assumption. If we are working with an order 2 HMM, we must make the bigram assumption, which states that the probability of a tag is only dependent on the previous tag, rather than the entire tag sequence. This can be expressed as

$$P(t_1 \ldots t_n) = \prod_{i=1}^{n} P(t_i \mid t_{i-1})$$

If we are working with an order 3 HMM, we have the trigram assumption, that is,

$$P(t_1 \ldots t_n) = \prod_{i=1}^{n} P(t_i \mid t_{i-1}, t_{i-2})$$

We can improve the performance of our model even more by using a classic natural language processing procedure called smoothing. Say we are interested in trigram HMMs; then we are interested in $P(t_i \mid t_{i-1}, t_{i-2})$. Instead of using the classic rules of conditional probability to solve this, we can instead use the following formulation

$$P(t_i \mid t_{i-1}, t_{i-2}) = \lambda_1 * P(t_i \mid t_{i-1}, t_{i-2}) + \lambda_2 * P(t_i \mid t_{i-1}) + \lambda_3 * P(t_i)$$

subject to

$$\lambda_1 + \lambda_2 + \lambda_3 = 1, \lambda_i \geq 0$$

The λ values are fit by an algorithm called linear interpolation, which is implemented in the below code. When we use smoothing, we can combat the overfitting that may occur when we use high order HMMs.

Without loss of generality, let's consider order 2 HMMs without smoothing. Using our new assumptions, we have

$$\{t^*\}_1^n = \arg\max_{t_1 \ldots t_n} P(w_1 \ldots w_n \mid t_1 \ldots t_n) P(t_1 \ldots t_n) = \arg\max_{t_1 \ldots t_n} \prod_{i=1}^{n} P(w_i \mid t_i) P(t_i \mid t_{i-1})$$

This formulation is serendipitously convenient; $P(w_i \mid t_i)$ corresponds to the emission probabilities, and $P(t_i \mid t_{i-1})$ corresponds to the transition probabilities. So now, how can we do this decoding? For

HMMs, we use the Viterbi algorithm, a dynamic programming algorithm for obtaining the maximum a posteriori (MAP) probability estimate of the most likely sequence of hidden states that results in a sequence of observed events. The Viterbi procedure for trigram HMMs is given in Algorithm 1, though note that the bigram Viterbi is very similar. The Viterbi algorithm begins by setting up a dynamic programming lattice, with one column for each observation (that is, word) and one row for each possible state. Each cell of the lattice, which is called $v_t(j)$ represents the probability that the HMM is in state j after seeing the first t observations and passing through the most probable state sequence $q_1 \mid q_{t-1}$. Tracing backwards through the lattice, we can recover the most probable sequence of parts of speech for each word in the given sentence.

What is the run time of the trigram Viterbi algorithm? Line 1 has a run time of K^2, since we have nested for loops, each of which iterates K times. The for loop on line 2 runs on the order of m time since the sequence is of length m. The for loop on line 3 runs on the order of K time, since we iterate through every state. The for loop on line 4 runs on the order of K time as well, since we iterate through every state. Finally, finding the max value for ℓ'' in lines 5 and 6 take on the order of K time since we must iterate through every state again. The block from line 2 to 6 has run time of $m * K^3$. Finally, the for loop on line 9 runs on the order of m time. Hence, the run time of the algorithm is $O(mK^3)$

Hyperparameter Selection

The main model parameter we could theoretically tune is the order of the HMM. Increasing the order of the model would mean the model takes into account more context for each word, but this could lead to overfitting. In practice, increasing the order of the model also increases the computation burden significantly, to the point where we would have to sit around and wait a very long time for the models to fit. For the purposes of this project, we will only work with order 2 and 3 HMMs and not tune this parameter.

Experimental Setup and Metrics

The data we will be using in this demonstration is a text file of sentences, containing over 1 million words in total. Each line has exactly 1 word and exactly 1 tag. We call this a large collection of text a ”corpus.” Each word is tagged with 1 of 36 parts of speech by a human tagger (presumably a grammar expert). These tags are taken to be the ground truth - though one should note there could be errors made by the human taggers. Our test set is a small set of natural text, untagged. We also have a version of the test set that is tagged by a human tagger that is in the same form as the training corpus. This will be used as our validation set to determine test accuracy. The metric of importance here will be test accuracy, that is, what percent of words in the test data are labelled correctly by the HMM.

The corpus files are read in by two python functions I wrote specifically to handle this type of data. The rest of the functions build the HMM, and the Analysis/Experiment Code section of the notebook runs the experiments.

We will have 4 experiment types. We will test the performance of trigram HHMs with smoothing, trigram HMMs without smoothing, bigram HMMs with smoothing, and bigram HMMs without smoothing. This setup will allow us to compare the performance of trigram HMMs and bigram HMMs as well as the effect of smoothing.

Here, we will explain the experiment for trigram HMMs with smoothing; all of the other experiments will proceed similarly. First, we we load in 1 percent of the training data, and train a trigram HMM on the data using smoothing. Then we will use this trained HMM to predict on the test data, and get a prediction accuracy score. Next, we will load in 5 percent of the entire training data, and train a trigram HMM on the data using smoothing. Again we will use this trained HMM to predict on the out of sample data, and get a prediction accuracy score. We repeat this for 10, 25, 50, 75, and 100 percent of the data. This will give us a sense of how increasing the amount of training data increases the accuracy of our model.

Experimental Results

Table 1 and Table 2 show the results of our experiments.

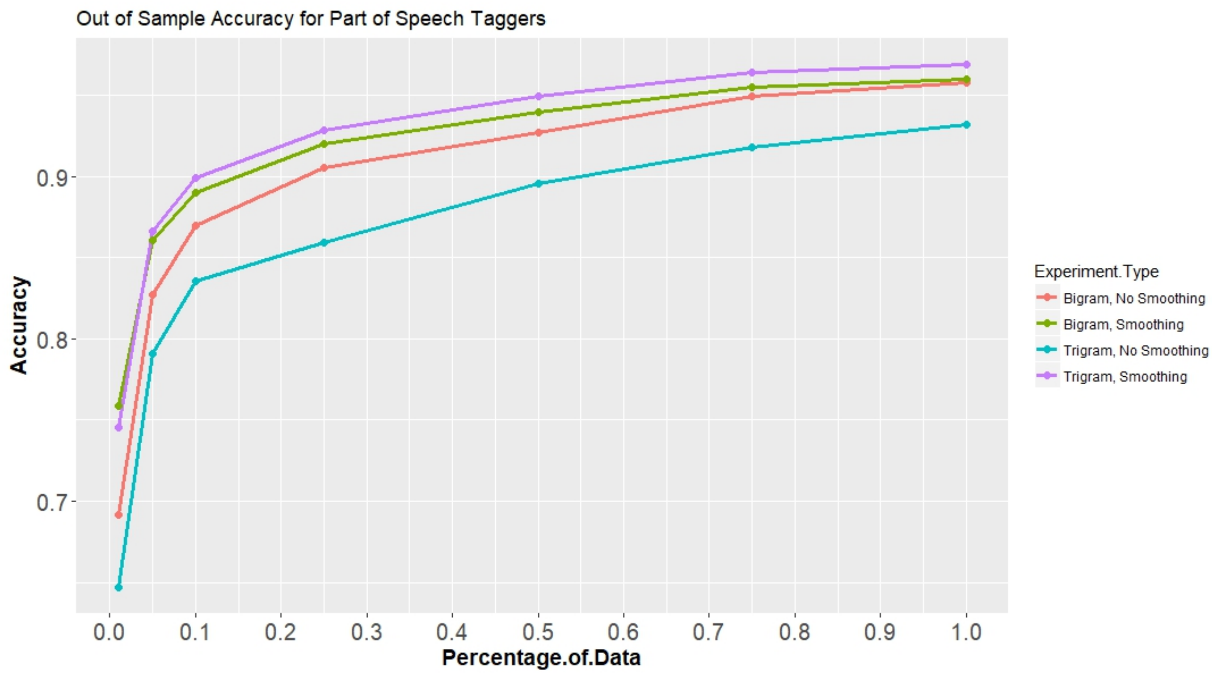
Table 1: Bigram HMM Results

Percent of Data	No Smoothing Accuracy	Smoothing Accuracy
1%	0.691	0.758
5%	0.827	0.860
10%	0.869	0.890
25%	0.905	0.919
50%	0.927	0.939
75%	0.949	0.954
100%	0.957	0.959

Table 2: Trigram HMM Results

Percent of Data	No Smoothing Accuracy	Smoothing Accuracy
1%	0.646	0.745
5%	0.790	0.866
10%	0.835	0.899
25%	0.859	0.928
50%	0.895	0.949
75%	0.917	0.964
100%	0.932	0.969

From Table 1, we see that the smoothed versions of the bigram HMM performs significantly better than its unsmoothed counterparts when the training data is small, but this advantage decreases as we use more training data. This points to the fact that bigram HMMs do not overfit to a high degree, so they benefit less from smoothing. When we use all of the training data, the bigram HMM achieves 96 percent accuracy, which is very impressive. Table 2 gives us the results for trigram HMMs. The effect of smoothing is much more apparent for trigram HMMs; smoothed trigram HMMs outperform their unsmoothed counterparts by a significant margin. Interestingly, the unsmoothed version of the bigram HMM actually achieves higher accuracies than the unsmoothed version of the trigram HMM. Still, the smoothed trigram HMM seems to have the best performance, achieving 97 percent accuracy when trained on the full data. This accuracy is very impressive; a trigram HMM could probably do better than the vast majority of literate English speakers. Finally, we see that as we increase the percentage of data on which we train each HMM, the accuracy improves, but the rate of increase in accuracy tapers off as the percent of the training data used increases. We can see these results in graphical form on the next page.



Errors and Mistakes

The most difficult part of this project was coding the Viterbi algorithm. I do not have much experience with dynamic programming, so learning this new technique took some time, but it was very rewarding. Implementing the linear interpolation to get the smoothing to work was also a challenge, but it helped me achieve better accuracies, so it was indeed worth it in the end.

References

[1] Michael Collins: Langauge Models.
http://www.cs.columbia.edu/mcollins/hmms-spring2013.pdf

[2] Michael Collins: Tagging Problems and Hidden Markov Models.
http://www.cs.columbia.edu/mcollins/hmms-spring2013.pdf

[3] Daniel Jurafsky & James H. Martin: Sequence Labeling for Parts of Speech and Named Entities
https://web.stanford.edu/jurafsky/slp3/8.pdf

[4] NLTK Project: Natural Language Toolkit Python Library
https://www.nltk.org

HMM

December 8, 2021

[1]:
```python
import math
import random
import numpy
from collections import *
```

[2]:
```python
class HMM:
    """
    Simple class to represent a Hidden Markov Model.
    """
    def __init__(self, order, initial_distribution, emission_matrix,␣
 ↪transition_matrix):
        self.order = order
        self.initial_distribution = initial_distribution
        self.emission_matrix = emission_matrix
        self.transition_matrix = transition_matrix
```

[3]:
```python
def read_pos_file(filename):
    """
    Parses an input tagged text file.
    Input:
    filename --- the file to parse
    Returns:
    The file represented as a list of tuples, where each tuple
    is of the form (word, POS-tag).
    A list of unique words found in the file.
    A list of unique POS tags found in the file.
    """
    file_representation = []
    unique_words = set()
    unique_tags = set()
    f = open(str(filename), "r")
    for line in f:
        if len(line) < 2 or len(line.split("/")) != 2:
            continue

        word = line.split("/")[0].replace(" ", "").replace("\t", "").strip()
        tag = line.split("/")[1].replace(" ", "").replace("\t", "").strip()
```

```
        file_representation.append( (word, tag) )
        unique_words.add(word)
        unique_tags.add(tag)

    f.close()

    return file_representation, unique_words, unique_tags
```

[4]:
```
def read_pos_file_modified(training_data_file):
    """
    A modified verysion of read_pos that only returns the file representation
    Input: training data file, a text file
    Output: The file represented as a list of tuples, where each tuple
    is of the form (word, POS-tag).
    """
    file_representation = []

    #open file
    f = open(str(training_data_file), "r")
    for line in f:
        if len(line) < 2 or len(line.split("/")) != 2:
            continue

        #split the string up
        word = line.split("/")[0].replace(" ", "").replace("\t", "").strip()
        tag = line.split("/")[1].replace(" ", "").replace("\t", "").strip()
        file_representation.append( (word, tag) )

    # close the file
    f.close()
    return file_representation
```

[5]:
```
################################### Testing RPOSFM
print (read_pos_file_modified("onesentence.txt"))
#expects The file represented as a list of tuples, where each tuple is of the
 ↪form (word, POS-tag).
#passes
```

```
[('The', 'DT'), ('New', 'NNP'), ('Deal', 'NNP'), ('was', 'VBD'), ('a', 'DT'),
('series', 'NN'), ('of', 'IN'), ('domestic', 'JJ'), ('programs', 'NNS'),
('enacted', 'VBN'), ('in', 'IN'), ('the', 'DT'), ('United', 'NNP'), ('States',
'NNPS'), ('between', 'IN'), ('1933', 'CD'), ('and', 'CC'), ('1936', 'CD'), (',',
','), ('and', 'CC'), ('a', 'DT'), ('few', 'JJ'), ('that', 'WDT'), ('came',
'VBD'), ('later', 'RB'), ('.', '.')]
```

[6]:
```
def parse_test_file(test_file):
    """
```

```
    Parses a test file into a list of lists, where the inner lists are sentences
    Input: A testing data file, test_file
    Outputs: a list of words in the file, and a list of lists as described above
    """

    # open the file
    f = open(test_file, "r")

    #split the file into a list
    list_of_words = f.read().split()
    testing_block = []
    L = len(list_of_words)
    count = 0

    while count < L:

        #container for each sentence
        sentence = []
        for word in list_of_words:

            #add word to the sentence
            sentence.append(word)
            count += 1
            if word == ".":

                # add sentence to the testing block
                testing_block.append(sentence)
                sentence = []

    return list_of_words, testing_block
```

[7]:

```
############################## TESTING PARSE ␣
↪####################################
# print (parse_test_file("testdata_untagged.txt"))
#expect the test data parsed into a list of list
#passes
```

[8]:

```
def wrangle_data(training_data_file, percent):
    """
    Inputs: training_data_file, a text file of tagged training data
            percent: a decimal represeting the amount of data you want to build␣
↪a model on
    Output: partitioned data, a sequence of (word, tag) tuples
    """
    #read in data
    data = read_pos_file_modified(training_data_file)
```

```
    #get the total length of the training data
    tokens = len(data)

    #get the amount of tokens you want to keep
    end_token = int(tokens*percent)

    partitioned_data = []

    #iterate until you read the end number of tokens
    for i in range(end_token):
        partitioned_data.append(data[i])

    return partitioned_data
```

[9]:
```
############################################ TESTING Wrangle Data␣
 ↪############################################
#print (wrangle_data("training.txt", 0.01))[1]
# expects 1 percent of the training data
# passes
```

[10]:
```
def get_unique_words_and_tags(data):
    """
    Gets the unique words and tags in a data set
    Input: Data in the representation returned by read pos
    Output: 2 sets, one that holds the unique words and one that holds the␣
 ↪unique tags
    """

    #init empty sets
    unique_words = set([])
    unique_tags = set([])


    for pair in data:

        #add the word if its not already been seen
        if pair[0] not in unique_words:
            unique_words.add(pair[0])

        #add the tag if its not already been seen
        if pair[1] not in unique_tags:
            unique_tags.add(pair[1])

    return unique_words, unique_tags
```

[11]:
```
########################## TESTING GET UNIQUE WORDS/TAGS␣
↪#########################
#fifty = wrangle_data("training.txt", 0.5)
#print (get_unique_words_and_tags(fifty))
# expects two sets with unique words and tags, on fifty percent of the training␣
↪data
# passes

fifty = wrangle_data("training.txt", 0.1)
#print (get_unique_words_and_tags(fifty))
# expects two sets with unique words and tags, on ten percent of the training␣
↪data
# passes
```

[12]:
```
def compute_counts(training_data, order):
    """
    Function that computes different relevant counts about the input file

    Input: Training data, a list of (word, POS-tag) pairs returned by the␣
↪function read_pos_file,
           Order, the order of the hidden markov model

    Output : If the HMM order is 2, the function returns a tuple consisting of:
                the number of tokens in training data
                dictionary that contains that contains C(ti,wi) for every␣
↪unique tag and unique word (keys correspond to tags)
                    The number of times word wi is tagged with ti
                a dictionary that contains  C(ti)
                    The number of times tag ti appears
                a dictionary that contains C(ti-1, ti)
                    The number of times the tag sequence ti-1, ti appears
             If the HMM order is 3, the function returns a tuple consisting of:
                the number of tokens in training data
                dictionary that contains that contains C(ti,wi) for every␣
↪unique tag and unique word (keys correspond to tags)
                a dictionary that contains  C(ti)
                a dictionary that contains  C(ti-1, ti)
                a dictionary that contains  C(ti-2, ti-1, ti)
                    The number of times the tag sequence ti-2, ti-1, ti appears


    """
    # get the number of tokens in the training set
    numtokens = len(training_data)

    # counts the number of times word wi is taked with tag ti
    word2tag_dict = defaultdict(lambda: defaultdict(int))
```

```
    # counts the number of times tag ti appears
    tag_count_dict = defaultdict(int)

    # counts the number of times the tag sequence ti-1, ti appears
    bigramdict = defaultdict(lambda: defaultdict(int))

    # counts the number of times the tag sequence ti-2, ti-1, ti appears
    trigramdict = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

    for i in range(0, numtokens):

        #increment the tag_count_dict
        tag_count_dict[training_data[i][1]] += 1

        #increment the word2tag_dict
        word2tag_dict[training_data[i][1]][training_data[i][0]]+=1

        if i > 0:

            if training_data[i-1][1] != ".":

                #increment the bigram dict
                bigramdict[training_data[i-1][1]][training_data[i][1]] += 1

        if order > 2:

            if i > 1:

                if (training_data[i-2][1] != ".") and (training_data[i-1][1] !=␣
↪"."):

                    #increment the trigram dict
                    ␣
↪trigramdict[training_data[i-2][1]][training_data[i-1][1]][training_data[i][1]]␣
↪+= 1



    if order == 2:
        return(numtokens, word2tag_dict, tag_count_dict, bigramdict)
    else:
        return(numtokens, word2tag_dict, tag_count_dict, bigramdict,␣
↪trigramdict)
```

[13]:
```
####### TESTING COMPUTE COUNTS #################
tinydata = read_pos_file('onesentence.txt')
```

```
#print(tinydata[0])
#Order 2 test on a single sentence
print(compute_counts(tinydata[0], 2)[0])
print(dict(compute_counts(tinydata[0], 2)[1]))
print(dict(compute_counts(tinydata[0], 2)[2]))



#expect a dictionary with the correct amount of words in the sentence
#passes

# Order 3 test on a single sentence
#print(dict(compute_counts(tinydata[0], 3)))
# expect a dictionary with the correct amount of words in the sentence
# passes
```

```
26
{'DT': defaultdict(<class 'int'>, {'The': 1, 'a': 2, 'the': 1}), 'NNP':
defaultdict(<class 'int'>, {'New': 1, 'Deal': 1, 'United': 1}), 'VBD':
defaultdict(<class 'int'>, {'was': 1, 'came': 1}), 'NN': defaultdict(<class
'int'>, {'series': 1}), 'IN': defaultdict(<class 'int'>, {'of': 1, 'in': 1,
'between': 1}), 'JJ': defaultdict(<class 'int'>, {'domestic': 1, 'few': 1}),
'NNS': defaultdict(<class 'int'>, {'programs': 1}), 'VBN': defaultdict(<class
'int'>, {'enacted': 1}), 'NNPS': defaultdict(<class 'int'>, {'States': 1}),
'CD': defaultdict(<class 'int'>, {'1933': 1, '1936': 1}), 'CC':
defaultdict(<class 'int'>, {'and': 2}), ',': defaultdict(<class 'int'>, {',':
1}), 'WDT': defaultdict(<class 'int'>, {'that': 1}), 'RB': defaultdict(<class
'int'>, {'later': 1}), '.': defaultdict(<class 'int'>, {'.': 1})}
{'DT': 4, 'NNP': 3, 'VBD': 2, 'NN': 1, 'IN': 3, 'JJ': 2, 'NNS': 1, 'VBN': 1,
'NNPS': 1, 'CD': 2, 'CC': 2, ',': 1, 'WDT': 1, 'RB': 1, '.': 1}
```

[14]:
```
def compute_initial_distribution(training_data, order):
    """
    Function that computes the intitial distributions of words, pi_1 and pi_2

    Input: Training data, a list of (word, POS-tag) pairs returned by the␣
↪function read_pos_file,
           Order, the order of the hidden markov model

    Output: If order = 2:
                Returns a one dim dictionary pi_1, that maps a tag to its␣
↪emission probability
            If order = 3:
                Returns a 2 dim dictionary pi_2 , that maps a bigram to its␣
↪emmission probabiiity

    """
```

```
    numtokens = len(training_data)

    # initialize the pi dictionaries
    pi_1 = defaultdict(int)
    pi_2 = defaultdict(lambda: defaultdict(int))

    if order==2:

        # the first tag is the second element of the first tuple
        first_tag = training_data[0][1]

        # increment the total count by 1
        pi_1[first_tag] += 1

    else:

        # access the first and second tags
        first_tag = training_data[0][1]
        second_tag = training_data[1][1]

        # increment the dictionary
        pi_2[first_tag][second_tag] += 1

    # set the total counts to be 1, we will normalize later
    order2count = 1
    order3count = 1

    for i in range(order -1, numtokens - order + 1):

        if order == 2:

            #if we encounter a period, we know we have the beginning of a sentence
            if training_data[i-1][1] == ".":

                # increment the count by 1
                pi_1[training_data[i][1]] += 1
                order2count += 1

        if order == 3:

            # if two words ago was a period, then we have a bigram that is at the begining of a word
            # will fail for 1 word sentences
            if training_data[i-2][1] == ".":
                pi_2[training_data[i-1][1]][training_data[i][1]] += 1
```

```
                order3count +=1

    if order == 2:

        for key, value in pi_1.items():

            #normalize by the order 2 count
            pi_1[key] = float(float(value)/float(order2count))

        return pi_1

    else:
        # iterate through the keys and values
        for key, value in pi_2.items():

            # value is a dict, whose "values" are counts
            for tag, count in value.items():

                #normalize by the order 3 count
                pi_2[key][tag] = float(float(count)/float(order3count))

        return pi_2
```

[15]:

```
######################## TESTING COMPUTE INTIAL DISTRIBUTION␣
↪###############################
littledata = read_pos_file('littledata.txt')
print(compute_initial_distribution(littledata[0], 3))
#expect DT-JJ to be 1/2 and NNP-NNPS to be 1/2
#passes

littledata = read_pos_file('littledata.txt')
print(compute_initial_distribution(littledata[0], 2))
# #expect DT to be 1/2 and NNP to be 1/2
# #passes
```

```
defaultdict(<function compute_initial_distribution.<locals>.<lambda> at
0x7fd19bed9700>, {'DT': defaultdict(<class 'int'>, {'JJ': 0.5}), 'NNP':
defaultdict(<class 'int'>, {'NNPS': 0.5})})
defaultdict(<class 'int'>, {'DT': 0.5, 'NNP': 0.5})
```

[16]:

```
def compute_emission_probabilities(unique_words, unique_tags, W, C):
    """
    Function computes the emmission matrix for different parts of speech given␣
↪training data

    Input: unique_words: a set of unique words returned by the read_pos function
           unique_tags : a set of unique tags returned by the read_pos function
```

```
        W : the C(ti, wi) dictionary returned by the function compute_counts
        C : the C(ti) dictionary returned by the function compute_counts

    Output: emission matrix, a 2d dict where the keys are parts of speech

    """

    #initialize the emission matrix
    emission_matrix = defaultdict(lambda: defaultdict(int))


    # tags are keys in the emission matrix
    for tag in unique_tags:
        for word in W[tag].keys():

                #caluculate emission prob
                emission_matrix[tag][word] = float(float(W[tag][word])/
↪float(C[tag]))

    return emission_matrix
```

[17]:

```
##################################### TESTING COMPUTE EMMISSION PROBABILITIES␣
↪######################
tinydata = read_pos_file('onesentence.txt')
unique_words2 = tinydata[1]
unique_tags2 = tinydata[2]
w1 = compute_counts(tinydata[0], 2)[1]
C1 = compute_counts(tinydata[0], 2)[2]
print(compute_emission_probabilities(unique_words2, unique_tags2, w1, C1))
# expect a dict with each word corresponding to its emmision prob
# passes

onethousand = read_pos_file("onethousandlines.txt")
unique_tags = onethousand[2]
unique_words = onethousand[1]
c1 = compute_counts(onethousand[0], 3)[1]
c2 = compute_counts(onethousand[0], 3)[2]
# print(compute_emission_probabilities(unique_words, unique_tags, c1, c2))
# # expect a dict with each word corresponding to its emmision prob
# # passes
```

```
defaultdict(<function compute_emission_probabilities.<locals>.<lambda> at
0x7fd1a441b160>, {'VBN': defaultdict(<class 'int'>, {'enacted': 1.0}), 'CD':
defaultdict(<class 'int'>, {'1933': 0.5, '1936': 0.5}), 'VBD':
defaultdict(<class 'int'>, {'was': 0.5, 'came': 0.5}), 'NN': defaultdict(<class
'int'>, {'series': 1.0}), 'JJ': defaultdict(<class 'int'>, {'domestic': 0.5,
'few': 0.5}), ',': defaultdict(<class 'int'>, {',': 1.0}), 'NNS':
```

```
defaultdict(<class 'int'>, {'programs': 1.0}), 'NNP': defaultdict(<class 'int'>,
{'New': 0.3333333333333333, 'Deal': 0.3333333333333333, 'United':
0.3333333333333333}), '.': defaultdict(<class 'int'>, {'.': 1.0}), 'CC':
defaultdict(<class 'int'>, {'and': 1.0}), 'IN': defaultdict(<class 'int'>,
{'of': 0.3333333333333333, 'in': 0.3333333333333333, 'between':
0.3333333333333333}), 'RB': defaultdict(<class 'int'>, {'later': 1.0}), 'DT':
defaultdict(<class 'int'>, {'The': 0.25, 'a': 0.5, 'the': 0.25}), 'WDT':
defaultdict(<class 'int'>, {'that': 1.0}), 'NNPS': defaultdict(<class 'int'>,
{'States': 1.0})})
```

[18]:
```python
def compute_lambdas(unique_tags, num_tokens, C1, C2, C3, order):
    """
    Function implements the Algorithm Compute_Lambdas

    Inputs: unique_tags: a set of unique tags returned by the read_pos function
            numtokens : number of words in the training corpus
            C1 : C(ti), The number of times tag ti appears
            C2: C(ti-1, ti), the Number of times the sequence ti -1, ti appears
            C3: C(ti-2, ti-1, ti) the number of times the sequence ti-2, ti-1,␣
 ↪ti appears

    Outputs: A list that contains lambda1, lambda2, lambda2

    """

    lambdas = [0.0,0.0,0.0]
    counter = 0
    # only if order is 3 do we consider trigrams
    if order == 3:

        # access ti -2
        for timinus2, value in C3.items():

            # access ti-1
            for timinus1, value2 in value.items():

                # access ti
                for ti, count in value2.items():

                    counter += 1

                    # initialize the argmax
                    argmax = 0
                    max_alpha = 0

                    # calculate the alpha scores
                    for i in range(3):
```

```
                        if i == 0:
                            if float(num_tokens) == 0:
                                alpha = 0
                            else:
                                alpha = float(float(C1[ti] - 1)/
↪float(num_tokens))

                        if i == 1:
                            if float(C1[timinus1] - 1) == 0:
                                alpha = 0
                            else:
                                #print "hey", C2[timinus1][ti]
                                #print C1[timinus1]
                                alpha = float(float(C2[timinus1][ti] - 1)/
↪float(C1[timinus1] - 1))

                        if i == 2:
                            if float(C2[timinus2][timinus1] - 1) == 0:
                                alpha = 0
                            else:
                                alpha = float(float(C3[timinus2][timinus1][ti]␣
↪- 1)/float(C2[timinus2][timinus1] - 1))

                        # find the biggest alpha
                        if alpha > max_alpha:
                            max_alpha = alpha
                            argmax = i

                    # increment alpha
                    lambdas[argmax] += C3[timinus2][timinus1][ti]

        # calculate the lambda values

        lambdas_sum = sum(lambdas)
        for i in range(order):
            lambdas[i] = float(float(lambdas[i])/float(lambdas_sum))

        return lambdas

    # if order is equal to 2
    else:

        #access ti- 1 and ti
        for timinus1, value2 in C2.items():
            for ti, count in value2.items():

                argmax = 0
```

```
                max_alpha = 0

                #calculate alpha scores
                for i in range(2):
                    if i == 0:
                        if float(num_tokens) == 0:
                            alpha = 0
                        else:
                            alpha = float(float(C1[ti] - 1)/float(num_tokens))
                    if i == 1:
                        if float(C1[timinus1] - 1) == 0:
                            alpha = 0
                        else:
                            alpha = float(float(C2[timinus1][ti] - 1)/
↪float(C1[timinus1] - 1))

                    if alpha > max_alpha:
                        max_alpha = alpha
                        argmax = i

                # increment lambda
                lambdas[argmax] += C2[timinus1][ti]

        # calculate the final lambda values
        lambdas_sum = sum(lambdas)

        for i in range(order):
            lambdas[i] = float(float(lambdas[i])/float(lambdas_sum))

        return lambdas
```

[19]:

```
######################################## TESTING COMPUTE LAMBDAS␣
↪#######################################################
alldata = read_pos_file("training.txt")
counts = compute_counts(alldata[0], 3)
unique_tags = alldata[2]
unique_words = alldata[1]
print(compute_lambdas(unique_tags, counts[0], counts[2], counts[3], counts[4],␣
↪2))
# expect 2 lambda values that are not 0 that sum to 1
# passes

tinydata = read_pos_file('onesentence.txt')
unique_tags = tinydata[2]
counts = compute_counts(tinydata[0], 3)
print(compute_lambdas(unique_tags, counts[0], counts[2], counts[3], counts[4],␣
↪3))
```

```
# expect 2 lambda values that are not 0 that sum to 1
# passes
```

```
[0.20472736536501893, 0.7952726346349811, 0.0]
[0.9583333333333334, 0.041666666666666664, 0.0]
```

[20]:
```
def compute_transition_matrix(training_data, unique_tags, order, use_smoothing):
    """
    Input: training_data, a file containing training data,
        unique tags, a set of unique tags in the data,
        order, the order of the HMM
        use smoothing: a boolean paramater

    Output: transistion matrix: a matrix that contains the transistion␣
↪probabiility between states
    """
    # order
    if order == 2:
        # get the counts
        # counts = compute_counts(training_data, 2)
        counts_trigram = compute_counts(training_data, 3)

        # get the number of tokens
        num_tokens = counts_trigram[0]

        # if smoothing is true, compute the appropriate lambda values
        if use_smoothing == True:
            lambdas = compute_lambdas(unique_tags, num_tokens,␣
↪counts_trigram[2], counts_trigram[3], counts_trigram[4], order)
        else:
            lambdas = [0, 1, 0]

        # compute the transition matrix
        transition_matrix = defaultdict(lambda: defaultdict(int))

        # obtain ti - 1
        for timinus1 in unique_tags:

            # obtain ti
            for ti in unique_tags:

                term1 =␣
↪(float(lambdas[1])*float(counts_trigram[3][timinus1][ti])/
↪float(counts_trigram[2][timinus1]))
                term2 = (float(lambdas[0])*float(counts_trigram[2][ti])/
↪float(num_tokens))
```

```
                transition_matrix[timinus1][ti] = float(term1) + float(term2)

        return transition_matrix


    else:

        #get your counts
        counts_trigram = compute_counts(training_data, 3)

        # get the number of tokens
        num_tokens = counts_trigram[0]

        # if smoothing is true, compute the appropriate lambda values
        if use_smoothing == True:
            lambdas = compute_lambdas(unique_tags, num_tokens, counts_trigram[2], counts_trigram[3], counts_trigram[4], order)
        else:
            lambdas = [0.0, 0.0, 1.0]

        # compute the transition matrix
        transition_matrix = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

        for timinus2 in unique_tags:

            # obtain ti - 1
            for timinus1 in unique_tags:

                # obtain ti
                for ti in unique_tags:

                    # calculate term 1 of the equation
                    if float(counts_trigram[3][timinus2][timinus1]) == 0:
                        term1 = 0
                    else:
                        term1 = (float(lambdas[2])*float(counts_trigram[4][timinus2][timinus1][ti]))/float(counts_trigram[3][timinus2][timinus1])

                    #caluculate term 2 of the equations
                    if float(counts_trigram[2][timinus1]) == 0:
                        term2 =0
                    else:
                        term2 = (float(lambdas[1])*float(counts_trigram[3][timinus1][ti])/float(counts_trigram[2][timinus1]))
```

```
                    # calculate term 3 of the equation
                    if float(num_tokens) == 0:
                        term3 =0
                    else:
                        term3 = (float(lambdas[0])*float(counts_trigram[2][ti])/
↪float(num_tokens))

                    #calculate the full probability
                    transition_matrix[timinus2][timinus1][ti] = term1 + term2 +␣
↪term3

        return transition_matrix
```

[21]:
```
################################################## TESTING COMPUTE TRANSISTION␣
↪MATRIX ##########################################################

tinydata = read_pos_file('onesentence.txt')
unique_tags = tinydata[2]
print(compute_transition_matrix(tinydata[0], unique_tags, 2, False))
# EXPECTS a 2d dict of transistion probs from one state to another
# passes

tinydata = read_pos_file('onesentence.txt')
unique_tags = tinydata[2]
#print(compute_transition_matrix(tinydata[0], unique_tags, 3, True))
# EXPECTS a 3d dict of transistion probs from two state to another
# passes
```

```
defaultdict(<function compute_transition_matrix.<locals>.<lambda> at
0x7fd1803628b0>, {'VBN': defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0,
'VBD': 0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0,
'CC': 0.0, 'IN': 1.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'CD':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.0, ',': 0.5, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC': 0.5, 'IN': 0.0, 'RB':
0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'VBD': defaultdict(<class 'int'>,
{'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0,
'NNP': 0.0, '.': 0.0, 'CC': 0.0, 'IN': 0.0, 'RB': 0.5, 'DT': 0.5, 'WDT': 0.0,
'NNPS': 0.0}), 'NN': defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD':
0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC':
0.0, 'IN': 1.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'JJ':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.0, ',': 0.0, 'NNS': 0.5, 'NNP': 0.0, '.': 0.0, 'CC': 0.0, 'IN': 0.0, 'RB':
0.0, 'DT': 0.0, 'WDT': 0.5, 'NNPS': 0.0}), ',': defaultdict(<class 'int'>,
{'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0,
'NNP': 0.0, '.': 0.0, 'CC': 1.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0,
'NNPS': 0.0}), 'NNS': defaultdict(<class 'int'>, {'VBN': 1.0, 'CD': 0.0, 'VBD':
```

```
0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC':
0.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'NNP':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD': 0.3333333333333333,
'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.3333333333333333, '.': 0.0,
'CC': 0.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS':
0.3333333333333333}), '.': defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0,
'VBD': 0.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0,
'CC': 0.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'CC':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.5, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC': 0.0, 'IN': 0.0, 'RB':
0.0, 'DT': 0.5, 'WDT': 0.0, 'NNPS': 0.0}), 'IN': defaultdict(<class 'int'>,
{'VBN': 0.0, 'CD': 0.3333333333333333, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.3333333333333333, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC': 0.0, 'IN':
0.0, 'RB': 0.0, 'DT': 0.3333333333333333, 'WDT': 0.0, 'NNPS': 0.0}), 'RB':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 1.0, 'CC': 0.0, 'IN': 0.0, 'RB':
0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'DT': defaultdict(<class 'int'>,
{'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.25, 'JJ': 0.25, ',': 0.0, 'NNS':
0.0, 'NNP': 0.5, '.': 0.0, 'CC': 0.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT':
0.0, 'NNPS': 0.0}), 'WDT': defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0,
'VBD': 1.0, 'NN': 0.0, 'JJ': 0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0,
'CC': 0.0, 'IN': 0.0, 'RB': 0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0}), 'NNPS':
defaultdict(<class 'int'>, {'VBN': 0.0, 'CD': 0.0, 'VBD': 0.0, 'NN': 0.0, 'JJ':
0.0, ',': 0.0, 'NNS': 0.0, 'NNP': 0.0, '.': 0.0, 'CC': 0.0, 'IN': 1.0, 'RB':
0.0, 'DT': 0.0, 'WDT': 0.0, 'NNPS': 0.0})})
```

[22]:

```python
def build_hmm(training_data, unique_tags, unique_words, order, use_smoothing):
    """
    Creates a fully trained Hidden Markov Model

    Inputs: training_data : a full training corpus,
            unique_tags : a set of parts of speech found in the training corpus
            unique_words : a set of words found in the training corpus
            order : the order of the markov chain
            Use_smoothing : a boolean parameter
    Outputs: a fully trained HMM object

    """

    # build an order 2 markov model

    counts = compute_counts(training_data, order)

    #compute the initial distribution
    initial_distribution = compute_initial_distribution(training_data, order)

    #compute the emission matrix
```

```
    W_dict = counts[1]
    C_dict = counts[2]
    emission_matrix = compute_emission_probabilities(unique_words, unique_tags,␣
↪W_dict, C_dict)

    #build a transition matrix
    transition_matrix = compute_transition_matrix(training_data, unique_tags,␣
↪order, use_smoothing)

    #build the hmm
    my_hmm = HMM(order, initial_distribution, emission_matrix,␣
↪transition_matrix)

    return my_hmm
```

[23]:

```
########################################## Testing Build HMM␣
↪################################################################

onethousand = read_pos_file("onethousandlines.txt")
unique_tags = onethousand[2]
unique_words = onethousand[1]
#print(build_hmm(onethousand[0], unique_tags, unique_words, 2, True))
#expect am HMM object
#passes

tinydata = read_pos_file('onesentence.txt')
unique_tags = tinydata[2]
unique_words = tinydata[1]
model = build_hmm(tinydata[0], unique_tags, unique_words, 3, False)
#Expect an HMM Object
#View the HMM Attributes
#print( "Order", model.order)
#print( "Intial Dist", model.initial_distribution)
#print( "Emmission Matrix", model.emission_matrix)
#print( "trans mat", model.transition_matrix)
# pass
```

[24]:

```
def update_hmm(hmm, sentence):
    """
    Function updates HMM based on new words it encounters
    Input: an hmm object and a sentence, a list of strings ending with a period
    Output: An updated hmm object
    """
    #get the attributes of the hidden markov model
    order = hmm.order
    initial_distribution = hmm.initial_distribution
    emission_matrix = hmm.emission_matrix
```

```
    transition_matrix = hmm.transition_matrix


    # get all the words
    unique_words = []

    for pos in emission_matrix:

        for word in emission_matrix[pos]:


            # add each word that the model has seen into unique words

            unique_words.append(word)

    #bool flag for if any new words were encountered
    new_word = False

    for word in sentence:

        # if the word is new, we want to add update the HMM
        if word not in unique_words:

            # if we get to this line, we have a new word
            new_word = True

            for part_of_speech in emission_matrix:

                for seen_word in emission_matrix[part_of_speech]:

                    # increment the each word by the same amount

                    emission_matrix[part_of_speech][seen_word] +=  0.00001

                # add the new word to each part of speech with a small␣
↪probability

                emission_matrix[part_of_speech][word] =  0.00001


    if new_word == True:

        #begin the normalization process
        for tag in emission_matrix:

            # find the normalizing term
            normalizer = sum(emission_matrix[tag].values())
```

```
            # normalize each word

            for finalword in emission_matrix[tag]:

                emission_matrix[tag][finalword] =␣
↪float(emission_matrix[tag][finalword])/float(normalizer)

        # updated_model = HMM(order, initial_distribution, emission_matrix,␣
↪transition_matrix)

    return HMM(order, initial_distribution, emission_matrix, transition_matrix)
```

[25]:
```
############################################################## Testing Update␣
↪HMM
alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 2, False)
print (hiddenmarkovmodel.order)
#expect a new hmm of order 2
#passes

alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 3, False)
print (hiddenmarkovmodel.order)
#expect a new hmm of order 3
#passes
```

```
2
3
```

[26]:
```
def log(number):
    """
    Caluculates the Logarithm of a number, and returns -inf in the number is 0
    Inputs : number, a real number
    Output : The log of a number, whihc is a real number of -inf
    """

    # log of 0 is negative infinity

    if number == 0:
        return float("-inf")
    else:
        return float(math.log(number))
```

```
################################# Testing LOG
print( log(0))
# Expect - inf
# passes

print( log(69))
# expect 4.2
#passes
```

```
-inf
4.23410650459726
```

[27]:
```
def bigram_viterbi(hmm, sentence):
    """
    Implements the Viterbi algorithm for the bigram model on an input HMM and a␣
↪sentence (a list of words and the period at the end).

    Input : HMM, an HMM object, and sentence, a list of words with a period at␣
↪the end
    Output: Tagged Words : a list of words and their tags
    """
    #get the data from the HMM object
    initial_distribution = hmm.initial_distribution
    emission_matrix = hmm.emission_matrix
    transition_matrix = hmm.transition_matrix


    # init V and BP
    V = defaultdict(lambda: defaultdict(int))
    bp = defaultdict(lambda: defaultdict(int))

    # length of the input sentence.
    L = len(sentence)

    # init the first column of the V matrix
    for pos in emission_matrix:

        #caluclate the first column of the matrix
        pi_sub_l = log(initial_distribution[pos])
        emissison_prob_x0 = log(emission_matrix[pos][sentence[0]])
        V[pos][0] = pi_sub_l + emissison_prob_x0

    for i in range(1, L):

        # l is a part of speech, more generally, a markov state
```

```
        for l in emission_matrix:

            max_prob = -float("inf")
            argmax = None

            # find the different l_prime values that can be take, then determine the argmax and the max
            for l_prime in emission_matrix:

                transition_probability = log(transition_matrix[l_prime][l])

                previous_prob = V[l_prime][i-1]
                possible_max = transition_probability + previous_prob

                #update l prime
                if possible_max > max_prob:
                    max_prob = possible_max
                    argmax = l_prime

            # if none
            if argmax == None:
                for l_prime in emission_matrix.keys():
                    if V[l_prime][i-1]>= max_prob:
                        max_prob = V[l_prime][i-1]
                        argmax = l_prime

            emission_prob = log(emission_matrix[l][sentence[i]])

            #update V and BP
            V[l][i] = emission_prob + max_prob
            bp[l][i] = argmax


    # begin traceback
    argmax2 = None
    max_val_holder = -float("inf")

    #access the first element in the Sequence
    for l_prime_pos in emission_matrix:
        v_mat_entry = V[l_prime_pos][L-1]

        #get highest entry
        if v_mat_entry > max_val_holder:
            max_val_holder = v_mat_entry
            argmax2 = l_prime_pos

    sentence[L-1] = (sentence[L-1], argmax2)
```

```
    #traceback
    for i in range(L-2, -1, -1):
        zi = bp[sentence[i+1][1]][i+1]
        sentence[i] = (sentence[i], zi)


    return sentence
```

[28]:
```
################################################################ TESTING BIGRAM␣
↪VITERBI

alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 2, False)
print (bigram_viterbi(hiddenmarkovmodel, ["My", "hips", "do", "not", "lie", ".
↪"]))
# Expect Possesive, noun, verb, negator, verb, period
# Passes

alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 2, False)
print (bigram_viterbi(hiddenmarkovmodel, ["I", "hope", "I", "pass", "this",␣
↪"class", "."]))
# Expect Pronoun, verb, Pronoun, verb, determiner, noun, perios
# Passes
```

```
[('My', 'PRP$'), ('hips', 'NNS'), ('do', 'VBP'), ('not', 'RB'), ('lie', 'VB'),
('.', '.')]
[('I', 'PRP'), ('hope', 'VBP'), ('I', 'PRP'), ('pass', 'VBP'), ('this', 'DT'),
('class', 'NN'), ('.', '.')]
```

[29]:
```
def trigram_viterbi(hmm, sentence):
    """
    Implements the Viterbi algorithm for the treigram model on an input HMM and␣
↪a sentence (a list of words and the period at the end).

    Input : HMM, an HMM object, and sentence, a list of words with a period at␣
↪the end
    Output: Tagged Words : a list of words and their tags
    """
    #get the data from the HMM object
    initial_distribution = hmm.initial_distribution
    emission_matrix = hmm.emission_matrix
```

```
    transition_matrix = hmm.transition_matrix

    # init V and BP
    V = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))
    bp = defaultdict(lambda: defaultdict(lambda: defaultdict(int)))

    # length of the input sentence.
    L = len(sentence)

    #being building the V matrix

    # init the l prime of the V matrix
    for pos_l_prime in emission_matrix:

        #init the l state
        for pos_l in emission_matrix:

            #caucluate the first column/plane in the 3d matrix

            pi_sub_lprime_l = log(initial_distribution[pos_l_prime][pos_l])
            emissison_prob_x0 = log(emission_matrix[pos_l_prime][sentence[0]])
            emissison_prob_x1 = log(emission_matrix[pos_l][sentence[1]])

            # put the entry in the matrix
            V[pos_l_prime][pos_l][1] = pi_sub_lprime_l + emissison_prob_x0 +␣
↪emissison_prob_x1

    for i in range(2, L):

        # l is a part of speech, more generally, a markov state
        for l_prime in emission_matrix:

            # find the different l_prime values that can be take, then␣
↪determine the argmax and the max
            for l in emission_matrix:

                bestmax = -float("inf")
                argmax = None

                for l_double_prime in emission_matrix:

                    #calculate possible entries

                    transition_probability =␣
↪log(transition_matrix[l_double_prime][l_prime][l])
                    previous_prob = V[l_double_prime][l_prime][i-1]
                    possible_max = transition_probability + previous_prob
```

```
                # update l double prime
                if possible_max > bestmax:
                    bestmax = possible_max
                    argmax = l_double_prime

            #if none in matrix
            if argmax == None:
                for l_double_prime in emission_matrix.keys():
                    if V[l_double_prime][l_prime][i-1] >= bestmax:
                        bestmax = V[l_double_prime][l_prime][i-1]
                        argmax = l_double_prime

            emission_prob = log(emission_matrix[l][sentence[i]])

            #update V and BP

            V[l_prime][l][i] = emission_prob + bestmax
            bp[l_prime][l][i] = argmax


# begin traceback
ZL_minus_1 = None
ZL_minus_2 = None
max_val_holder= -float("inf")

#access the first element in the Sequence
for state_1 in emission_matrix:
    for state_2 in emission_matrix:
        v_mat_entry = V[state_1][state_2][L-1]

        #get highest entry
        if v_mat_entry > max_val_holder:
            max_val_holder = v_mat_entry
            ZL_minus_1 = state_2
            ZL_minus_2 = state_1

#init last Z values
sentence[L-1] = (sentence[L-1], ZL_minus_1)
sentence[L-2] = (sentence[L-2], ZL_minus_2)

#traceback
for i in range(L-3, -1, -1):
    zi = bp[sentence[i+1][1]][sentence[i+2][1]][i+2]
    sentence[i] = (sentence[i], zi)

return sentence
```

[30]:
```
########################################## TESTING TRIGRAM

alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 3, False)
print (trigram_viterbi(hiddenmarkovmodel, ["I", "am", "the", "machine", "."]))
# expecg Pronoun, verb, det, noun, period
#passes

alldata = read_pos_file("training.txt")
unique_tags = alldata[2]
unique_words = alldata[1]
hiddenmarkovmodel = build_hmm(alldata[0], unique_tags, unique_words, 3, False)
print (trigram_viterbi(hiddenmarkovmodel, ["I", "am", "happy", "."]))
# expecg Pronoun, verb, adj, period
#passes
```

```
[('I', 'PRP'), ('am', 'VBP'), ('the', 'DT'), ('machine', 'NN'), ('.', '.')]
[('I', 'PRP'), ('am', 'VBP'), ('happy', 'JJ'), ('.', '.')]
```

0.1 ANALYSIS CODE

[31]:
```
def compute_accuracy(tagged_data, results):
    '''
    Computes similarity between the actual tagged data and the algorithm's tags
    Input: tagged data, a list of tuples, and results, a list of tuples
    '''

    #init a numerator and a denominator
    numerator = 0
    denominator = 0
    for i in range(len(tagged_data)):

            #add a denominator regardless
            denominator += 1
            if tagged_data[i] == results[i]:

                #if we get a match, increment the numerator
                numerator += 1

    #calculate the percent accuracy
    percent_accurate = float(numerator)/float(denominator)

    return percent_accurate
```

```
[32]: ############################################ TESTING COMPUTE ACCURACY␣
      ↪############################
      a = ["a", "a", "a", "a"]
      b = ["a", "b", "a", "a"]
      print (compute_accuracy(a,b))
      # expect .75
      #passes

      a = ["a", "a", "a", "a"]
      b = ["a", "a", "a", "a"]
      print (compute_accuracy(a,b))
      # expect 1.0
      #passes
```

```
0.75
1.0
```

```
[33]: def bigram_validate(training_data, percent, testdata_untagged, testdata_tagged,␣
      ↪order, use_smoothing):
          """
          Computes the out of sample accuracy of the POS tagging algorithm for bigram␣
      ↪HMM
          Inputs: a file training data, a percentage of data, untagged test data,␣
      ↪tagged test data, the order of the markov chain,
                  a boolean parameter use smoothing

          Output: Accuracy, a real number between 0 and 1
          """

          #wrangle the data
          my_data = wrangle_data(training_data, percent)
          unique_words, unique_tags = get_unique_words_and_tags(my_data)

          #build an HMM
          old_hmm = build_hmm(my_data, unique_tags, unique_words, order,␣
      ↪use_smoothing)
          list_of_words, test_data_parsed = parse_test_file(testdata_untagged)

          #update the HMM if need be
          new_hmm = update_hmm(old_hmm, list_of_words)

          # put the predidted tags into a master list
          full_results = []
          for sentence in test_data_parsed:
              results = bigram_viterbi(new_hmm, sentence)
              for tup in results:
                  full_results.append(tup)
```

```
    # read in the validation data
    validation_data = read_pos_file_modified(testdata_tagged)


    #get the final accuracy
    return compute_accuracy(validation_data, full_results)
```

[34]:
```
######################################## TESTING BIGRAM Validate␣
 ↪######################################
print (bigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, True))
print (bigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, False))

#expect two decimals between 0-1 with the first value higher than the second
# Passes
```

```
0.7587268993839835
0.6919917864476386
```

[35]:
```
def trigram_validate(training_data, percent, testdata_untagged,␣
 ↪testdata_tagged, order, use_smoothing):
    """
    Computes the out of sample accuracy of the POS tagging algorithm for bigram␣
 ↪HMM
    Inputs: a file training data, a percentage of data, untagged test data,␣
 ↪tagged test data,
            the order of the markov chain, a boolean parameter use smoothing

    Output: Accuracy, a real number between 0 and 1
    """
    #wrangle the data
    my_data = wrangle_data(training_data, percent)
    unique_words, unique_tags = get_unique_words_and_tags(my_data)

    #build an hmm
    old_hmm = build_hmm(my_data, unique_tags, unique_words, order,␣
 ↪use_smoothing)
    list_of_words, test_data_parsed = parse_test_file(testdata_untagged)

    #update the HMM if any new words are encountered
    new_hmm = update_hmm(old_hmm, list_of_words)

    # put the algorithm's predictions into a master list
    full_results = []
    for sentence in test_data_parsed:
```

```
        results = trigram_viterbi(new_hmm, sentence)
        for tup in results:
            full_results.append(tup)

    #read in the validation data
    validation_data = read_pos_file_modified(testdata_tagged)

    #get an accuracy value
    return compute_accuracy(validation_data, full_results)
```

[36]:
```
#################################### TESTING TRIGRAM VALIDATE
print (trigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 3, True))
print (trigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 3, False))

# expect two decimals between 0-1 with the first value higher than the second
# Passes
```

```
0.7453798767967146
0.6468172484599589
```

0.2 EXPERIMENT CODE

[37]:
```
################################################## EXPERIMENT ONE␣
 ↪#####################################################

'''
In experiment one, we build seven bigram HMMs on the first 1%, 5%, 10%, 25%,␣
 ↪50%,
75%, and 100% of the training corpus without smoothing and obtain 7 accuracy␣
 ↪values
'''
onepercent_1 = bigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, False)
fivepercent_1 = bigram_validate("training.txt", 0.05, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, False)
tenpercent_1 = bigram_validate("training.txt", 0.1, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, False)
twentyfivepercent_1 = bigram_validate("training.txt", 0.25, "testdata_untagged.
 ↪txt", "testdata_tagged.txt", 2, False)
fiftypercent_1 = bigram_validate("training.txt", 0.5, "testdata_untagged.txt",␣
 ↪"testdata_tagged.txt", 2, False)
seventyfivepercent_1 = bigram_validate("training.txt", 0.75, "testdata_untagged.
 ↪txt", "testdata_tagged.txt", 2, False)
```

```
onehundopercent_1 = bigram_validate("training.txt", 1, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, False)
experiment_1_results = [onepercent_1, fivepercent_1, tenpercent_1,
                        twentyfivepercent_1, fiftypercent_1,␣
↪seventyfivepercent_1, onehundopercent_1]
print(experiment_1_results)

"""
[0.6919917864476386, 0.8275154004106776, 0.8696098562628337, 0.
↪9055441478439425, 0.9271047227926078, 0.9496919917864476, 0.9579055441478439]
"""
```

```
[0.6919917864476386, 0.8275154004106776, 0.8696098562628337, 0.9055441478439425,
0.9271047227926078, 0.9496919917864476, 0.9579055441478439]
```

[37]:
```
'\n[0.6919917864476386, 0.8275154004106776, 0.8696098562628337,
0.9055441478439425, 0.9271047227926078, 0.9496919917864476,
0.9579055441478439]\n'
```

[38]:
```
################################################# EXPERIMENT TWO␣
↪#######################################################

'''
In experiment 2, we build 7 trigram HMMs on the first 1%, 5%, 10%, 25%, 50%,
75%, and 100% of the training corpus without smoothing and obtain 7 accuracy␣
↪values
'''
onepercent_2 = trigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, False)
fivepercent_2 = trigram_validate("training.txt", 0.05, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, False)
tenpercent_2 = trigram_validate("training.txt", 0.1, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, False)
twentyfivepercent_2 = trigram_validate("training.txt", 0.25, "testdata_untagged.
↪txt", "testdata_tagged.txt", 3, False)
fiftypercent_2 = trigram_validate("training.txt", 0.5, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, False)
seventyfivepercent_2 = trigram_validate("training.txt", 0.75,␣
↪"testdata_untagged.txt", "testdata_tagged.txt", 3, False)
onehundopercent_2 = trigram_validate("training.txt", 1, "testdata_untagged.
↪txt", "testdata_tagged.txt", 3, False)
experiment_2_results = [onepercent_2, fivepercent_2, tenpercent_2,␣
↪twentyfivepercent_2, fiftypercent_2, seventyfivepercent_2, onehundopercent_2]
print(experiment_2_results)

'''
```

```
[0.6468172484599589, 0.7905544147843943, 0.8357289527720739, 0.
↪8593429158110883, 0.8952772073921971, 0.917864476386037, 0.9322381930184805]

'''
```

[0.6468172484599589, 0.7905544147843943, 0.8357289527720739, 0.8593429158110883, 0.8952772073921971, 0.917864476386037, 0.9322381930184805]

[38]: '\n[0.6468172484599589, 0.7905544147843943, 0.8357289527720739, 0.8593429158110883, 0.8952772073921971, 0.917864476386037, 0.9322381930184805]\n\n'

[39]:
```
################################################ EXPERIMENT 3
'''
In experiment three, we build seven bigram HMMs on the first 1%, 5%, 10%, 25%,␣
↪50%,
75%, and 100% of the training corpus with smoothing and obtain 7 accuracy values
'''
onepercent_3 = bigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, True)
fivepercent_3 = bigram_validate("training.txt", 0.05, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, True)
tenpercent_3 = bigram_validate("training.txt", 0.1, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, True)
twentyfivepercent_3 = bigram_validate("training.txt", 0.25, "testdata_untagged.
↪txt", "testdata_tagged.txt", 2, True)
fiftypercent_3 = bigram_validate("training.txt", 0.5, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, True)
seventyfivepercent_3 = bigram_validate("training.txt", 0.75, "testdata_untagged.
↪txt", "testdata_tagged.txt", 2, True)
onehundopercent_3 = bigram_validate("training.txt", 1, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 2, True)
experiment_3_results = [onepercent_3, fivepercent_3, tenpercent_3,␣
↪twentyfivepercent_3, fiftypercent_3, seventyfivepercent_3, onehundopercent_3]
print (experiment_3_results)

'''
[0.7587268993839835, 0.8603696098562629, 0.8901437371663244, 0.919917864476386,␣
↪0.9394250513347022, 0.9548254620123203, 0.9599589322381931]

'''
```

[0.7587268993839835, 0.8603696098562629, 0.8901437371663244, 0.919917864476386, 0.9394250513347022, 0.9548254620123203, 0.9599589322381931]

[39]: '\n[0.7587268993839835, 0.8603696098562629, 0.8901437371663244, 0.919917864476386, 0.9394250513347022, 0.9548254620123203, 0.9599589322381931]\n\n'

[40]:
```
######################################################## EXPERIMENT 4
'''
In experiment 2, we build 7 trigram HMMs on the first 1%, 5%, 10%, 25%, 50%,
75%, and 100% of the training corpus with smoothing and obtain 7 accuracy values
'''

onepercent_4 = trigram_validate("training.txt", 0.01, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, True)
fivepercent_4 = trigram_validate("training.txt", 0.05, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, True)
tenpercent_4 = trigram_validate("training.txt", 0.1, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, True)
twentyfivepercent_4 = trigram_validate("training.txt", 0.25, "testdata_untagged.
↪txt", "testdata_tagged.txt", 3, True)
fiftypercent_4 = trigram_validate("training.txt", 0.5, "testdata_untagged.txt",␣
↪"testdata_tagged.txt", 3, True)
seventyfivepercent_4 = trigram_validate("training.txt", 0.75,␣
↪"testdata_untagged.txt", "testdata_tagged.txt", 3, True)
onehundopercent_4 = trigram_validate("training.txt", 1, "testdata_untagged.
↪txt", "testdata_tagged.txt", 3, True)
experiment_4_results = [onepercent_4, fivepercent_4, tenpercent_4,␣
↪twentyfivepercent_4, fiftypercent_4, seventyfivepercent_4, onehundopercent_4]
print(experiment_4_results)

'''
[0.7453798767967146, 0.86652977412731, 0.8993839835728953, 0.9281314168377823,␣
↪0.9496919917864476, 0.9640657084188912, 0.9691991786447639]
'''
```

[0.7453798767967146, 0.86652977412731, 0.8993839835728953, 0.9281314168377823, 0.9496919917864476, 0.9640657084188912, 0.9691991786447639]

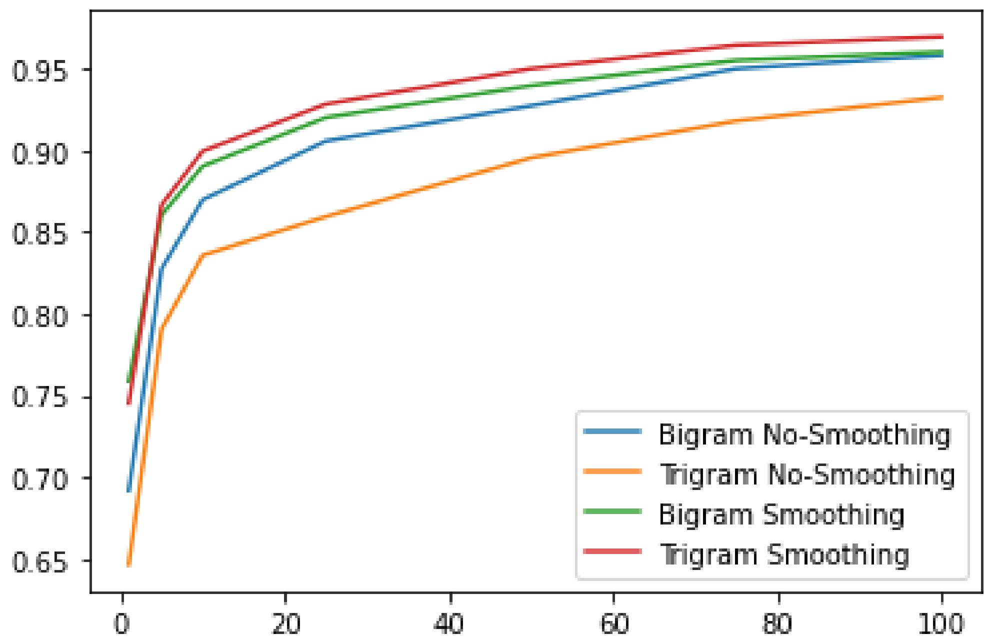
[40]: '\n[0.7453798767967146, 0.86652977412731, 0.8993839835728953, 0.9281314168377823, 0.9496919917864476, 0.9640657084188912, 0.9691991786447639]\n\n'

[41]:
```
# importing package
import matplotlib.pyplot as plt

# create data
x = [1,5,10,25,50, 75, 100]
y1 = experiment_1_results
```

```
y2 = experiment_2_results
y3 = experiment_3_results
y4 = experiment_4_results
# plot lines
plt.plot(x, y1, label = "Bigram No-Smoothing")
plt.plot(x, y2, label = "Trigram No-Smoothing")
plt.plot(x, y3, label = "Bigram Smoothing")
plt.plot(x, y4, label = "Trigram Smoothing")

plt.legend()
plt.show()
```



[ ]: