Soham Shah

Intro to AI

Rhonda Hoenigman

# Assignment 8: HMM

## Purpose:

The purpose of this assignment was to create a Hidden Markov Model with a training dataset and then use that dataset and the Viterbi algorithm to predict the tag for each word in a given sentence.

## Procedure:

## Transition Probability:

To generate transition probabilities, I used a combination of dictionaries and dictionary comprehensions.

There are two essential pieces of data I needed to get in order to figure out the transition probability. The first is the probability of one tag given a previous tag and the other is the number of times a given tag occurred.

The first thing I did was create a dictionary for the tags. And when parsing the penntree.tag file count the occurrences of each tag.

tags = {}

**with** open(**"penntree.tag"**, **"r"**) **as** sentences:

**for** line **in** sentences:

tag = line.split(**"\t"**)[1].rstrip(**'\n'**)

**if** tag **not in** tags:  
 tags[tag] = 1  
**else**:  
 tags[tag] += 1

This code is responsible for parsing the file and finding each type of tag and counting how many times it occurs.

The next thing I needed to do was count how many two tags in a row appeared. This gives the count of (tag1 and tag2 )

In order to do this I used a dictionary of dictionaries again.

transitions = {key: {key2: 0 **for** key2 **in** tags} **for** key **in** tags}

**for** i **in** lines:  
 splitLine = i.split()  
 lenSplitLine = len(splitLine)  
 **if** (lenSplitLine > 3):  
 prev = splitLine[1]  
 j = 3  
 **while** (j < lenSplitLine):  
 transitions[splitLine[j]][prev]+=1  
 prev = splitLine[j]  
 j += 2

Lines was the place where I stored all the sentences. I didn’t use “SSSS” and “EEEE” to tell if something was the start or end of a sentence. Instead, I just appended each sentence to the array of lines after combining all the words in the sentence. This way I avoided having to tell if there was a sentence at all. Because of this approach, I just had to split the string that was the sentence and take every second word as that was a tag. I created the iterator ‘j’ and incremented it by two in the while loop to iterate over the sentence. The dictionary transitions held a dictionary of all the tags and each key had a dictionary of tags with the count 0. Each time I encountered a set of tags I would just increment the corresponding transition value by one.

Now that these two pieces of data were assembled, I just used a dictionary comprehension to take both dictionaries and get the relevant probability.

transition\_probability = {key: {key2: transitions[key][key2]/tags[key2] **for** key2 **in** tags} **for** key **in** tags}

transition\_probability is a dictionary of dictionaries where the two keys are the sequence of tags and the value is the probability that it occurs.

## Emission Probabilities

Creating the emission probabilities followed a similar process to transition probabilities.

emissions = {key: {} **for** key **in** tags}  
**for** i **in** lines:  
 splitLine = i.split()  
 lenSplitLine = len(splitLine)  
 **if** (lenSplitLine > 1):  
 **for** i **in** range(0,lenSplitLine,2):  
 word = splitLine[i]  
 tag = splitLine[i+1]  
 **if** (word **in** emissions[tag]):  
 emissions[tag][word] += 1  
 **else**:  
 emissions[tag][word] = 1

Here’s the code. Once again, I created a dictionary of dictionaries which held the data. The first dictionary was the tags. The second dictionary contained the word and the value was the number of times that combination occurred. The above code shows how I parse the lines array to generate these counts.

At the count of the tags themselves was already done previously I could just move on to creating the emission probability dictionary. To do this, I used a dictionary comprehension again. These were very convenient through the project in creating dictionaries with very little wasted code.

observations = (**'This'**, **'is'**, **'a'**, **'sentence'**, **'.'**)

emission\_probability = {tag: {word: emissions[tag][word] / tags[tag] **if** word **in** emissions[tag] **else** 0 **for** word **in** observations} **for** tag **in** tags}

There are a couple important things to nore with the generation of the emission\_probability dictionary. The first key is the tags. The second key is the words. However, it seemed excessive to generate a dictionary for every possible word in the penntree.tag file. So, I only used the words that were in the sentence we were passing in. It takes the words in observations (the sentence) and then looks them up in the emissions dictionary I created earlier. It then divides by the amount of times that tag is in the penntree file. As a word cannot be all tags, if that word isn’t in emissions[tag] then I would just set that probability to 0.

## Viterbi Algorithm

The final thing necessary was passing in all information to the Viterbi algorithm. The algorithm allows you to find the most probable path for a given set of observations. It takes the observations and then uses the transition and emission probabilities in order to calculate the most likely path of states for the observations. In the case of this implementation, an array of dictionaites V = [{}] is used to store the back probabilities.

## Data

The data was taken from the penntree.tag file. This is a series of sentences where linguists have gone through and tagged each word and what it corresponds to. This is useful because we can use it as training data for the computer to create a model. The sentences are all broken up by newlines and the word and tag are delineated by tabs. I wanted to get each sentence into the array as a single string so that it was easier to work with. To do this, I used a temp array where I could store stuff, combine it, and then place that into the array of sentences. Here’s the code I used to process the data:

lines = []  
temparray = []

**with** open(**"penntree.tag"**, **"r"**) **as** sentences:  
 **for** line **in** sentences:  
 **if** (line==**"\n"**):  
 lines.append(**" "**.join(temparray))  
 temparray = []  
 **else**:  
 temparray.append(line.rstrip())

As you can see, if the code encounters a new line, it joins everything in the temp array and appends it to lines. So, the complete sentence does to lines as a single string. Otherwise, it takes the string and appends it to a temporary array as it’s a word and tag of a sentence.

## Result

Overall, the results were interesting. For the shorter strings I got the right answer.

Given the input:

observations = ('This', 'is', 'a', 'sentence', '.')

the result was “DT VBZ DT NN .” which is the correct answer.

Given the input:

observations = ('Can', 'a', 'can', 'can', 'a', 'can', '?')

The code gave “MD DT MD NN DT MD .” which is the right answer as well. As written in the assignment, the first and third Can’s are MD and NN. Essentially, depending on the context, the program gives a different output.

However, when given the input:

observations = ('This', 'might', 'produce', 'a', 'result', 'if', 'the', 'system', 'works', 'well', '.')

it results in the output:

“DT MD NN DT NN IN DT NN VBZ RB .”

When it should be:

“DT MD VB DT NN IN DT NN VBZ RB .”

As you can see, while most of the words match, the third word has a different tag. I believe this has to do with rounding and the way probabilities are being calculated in my program. The Viterbi algorithm states: “The steps of states are DT MD NN DT NN IN DT NN VBZ RB . with highest probability of 3.555498643462909e-42.” Essentially, the probabilities are becoming increasingly small which leads to rounding vs truncating errors. Another issue is that it depends on context similar to the question above.

Here is the rest of the sentences and the output for each.

Input: observations = ('Can', 'a', 'can', 'move', 'a', 'can', '?')

Output: MD DT MD NN DT MD .

Expected: 'MD', 'DT', 'MD', 'VB', 'DT', 'NN', '.'

Input: observations = ('Can', 'you', 'walk', 'the', 'walk', 'and', 'talk', 'the', 'talk', '?')

Output: MD PRP VBP DT VB CC NN DT VB

Expected: 'MD', 'PRP', 'VBP', 'DT', 'NN', 'CC', 'VB', 'DT', 'NN', '.'

For some reason, NN is VB and VB is NN. It appears that there’s something wrong with the way NN is being classified.