English POS Tagger

Group 5

What is Part-of-Speech

In any language there is a set of accepted rules that its speakers and writers unknowingly follow to construct coherent and structured sentences, that is we just don't spew out random words on spot to create a sentence while in a discussion or when writing which will automatically make sense.

For example in the english language one general rule is that we most commonly put adjectives before a noun(red flower , bright candle). These rules are related to syntax which is a set of rules, principles and processes to form structured sentences.

To depict these rules, words were assigned to different classes which were defined upon the roles assumed by these words in a given phrase. These roles are what we call “PART-OF-SPEECH”(POS for short).

When and Why to use POS tagging

POS tagging is done to assign different tags/labels to words which identifies the words part of speech. For example :-

For the word “living” the POS tag will be different for different sentences like

1. “Living being”, “living room”, here living is an adjective.
2. “He does it for a living ”, here living is a noun.
3. “He has been living here”, here living is a verb

So POS tagging is essentially used for normalizing text in a more intelligent manner or to extract information based on the words’s POS tag.

How to do POS Tagging

There are four main pos tagging methods :-

1. Manual tagging :-

   As the name implies this is done by humans, this is very slow, very much error prone, depends on the level of semantic knowledge of the person for that particular language.

2. Rule based tagging :-

   This consists of a series of rules (for i.e : if the word preceding it is an article and the word succeeding it is a noun, then its an adjective...) these rules are used by computers for pos tagging.

3. Stochastic / probabilistic method :-

   Assigning a POS tag according to the probability that a word belongs to a certain tag. Among these methods there are more methods one is the discriminative probabilistic classifiers(Ex logistic regression, conditional random fields) and generative probabilistic classifiers(Ex hidden markov model). So far this is the best method which achieved the highest accuracy amongst the other methods.

4. Deep learning method :-

   Here advanced machine learning algorithms are used to identify and infer pos tags. Compared to the 3rd method this method produces more or less at the same accuracy level of the probabilistic method, but its creation and functioning is very complex and time consuming.

Stochastic Part-of-Speech Tagging

The term ‘Stochastic tagger’ can refer to any model which incorporates frequency or probability as the backbone of the model. The simplest approach could be to judge a word solely based on the probability that it occurs with a particular tag. In other words, the tag associated most frequently in the training set with the word is the one assigned to an ambiguous instance of the word. The problem with this approach is that it may generate valid tags in some cases, it can also yield inadmissible sequences of tags.

An alternative to the above approach is the *n-gram* approach, referring to the fact that the best tag for a given word is determined by the probability that it occurs with n previous tags.

We have combined the above two approaches, using both tag sequence probabilities and word frequency measurements. This is known as HIdden Markov Model (HMM).

Markov Model

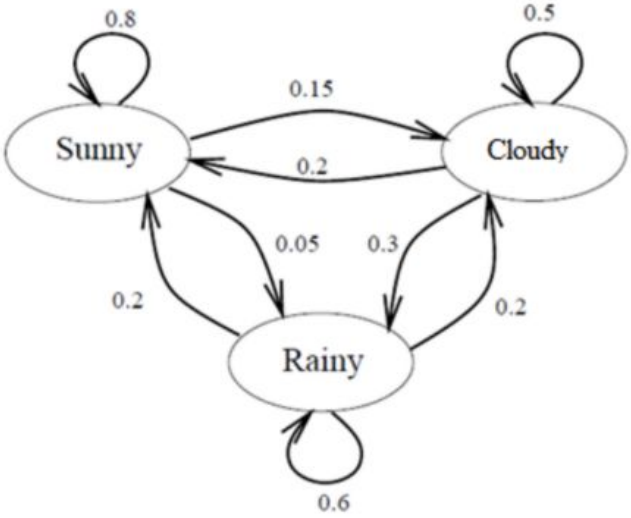
To explain the Markov Model, we consider a case of weather forecast. Let's say we have a data set of observations of the weather conditions on a particular day. There are only three kinds of weather conditions, namely

- Rainy
- Sunny
- Cloudy

Suppose we have a sequence of weather conditions on days like this:

Sunny, Rainy, Cloudy, Cloudy, Sunny, Sunny, Sunny, Rainy

Now using the data we have, we can construct the following state diagram with the labelled probabilities.



$$P(Sunny|Sunny) = 0.8$$
$$P(Rainy|Sunny) = 0.05$$
$$P(Cloudy|Sunny) = 0.15$$
$$\Big\} 1$$

$$P(Sunny|Rainy) = 0.2$$
$$P(Rainy|Rainy) = 0.6$$
$$P(Cloudyy|Rainy) = 0.2$$
$$\Big\} 1$$

$$P(Sunny|Cloudy) = 0.2$$
$$P(Rainy|Cloudy) = 0.3$$
$$P(Cloudy|Cloudy) = 0.5$$
$$\Big\} 1$$

In order to compute the probability of today's weather given N previous observations, we can use the following Markovian property:

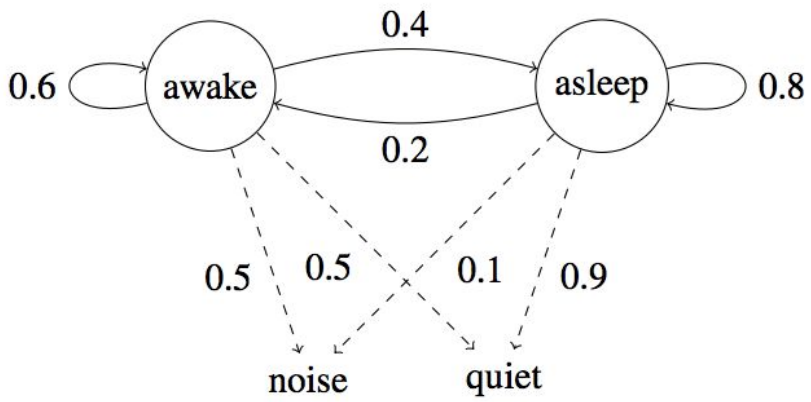
$$P(q_1, \dots, q_n) = \prod_{i=1}^{n} P(q_i \mid q_{i-1})$$

The Markov property suggests that the distribution for a random variable in the future depends solely on its distribution in the current state, and none of the previous states have any impact on the future states. It is clearly visible that the probability of a day being *Cloudy* depends solely on its previous day's weather and not any other days in the past.

Hidden Markov Model (HMM)

Let us again take an example of a child sleeping inside a room and there is a caretaker for the child. As a caretaker, the most important task is to make the child go to sleep and make sure he is sound asleep. Once the child has been made to sleep, it is to be made sure that the child is actually asleep and not up to some mischief. But, the caretaker cannot enter the room without waking up the child. So, the caretaker has to decide it through the noises coming out of the room. The noises are the states here (either **quiet** or **noisy**).

We have the following state diagram for the child's behaviour:



There are two kinds of probabilities in the state diagram:

- One is **emission** probability, which represents the probabilities of making certain observations given a particular state. This is denoted by ‘e’ in the code.
- The other ones are **transition** probabilities, which represent the probabilities of transitioning to other states given a particular state. This is denoted by ‘q’ in the code.

The Markovian property is applicable to the following model also. But, the flaw with Markov property is that if the child has been awake for 5 minutes, then the probability of going to sleep is much lower than that if he had been awake for 1 hour. Therefore, history does matter.

Our problem is that we have an initial state, but we do not have any set of states. If we had the states, we could calculate the probability of the sequence. All we have is a set of observations made. That is why, this model is called **Hidden** Markov Model - the actual states over time are hidden.

HMMs for Part-of-Speech Tagging

For HMM, we know that we need a set of observations and a set of possible states. In the POS tagging problem, we have the words in the sentence as the observation. As for the *hidden* states, that would be the POS tags for the words.

The **transition probabilities** would be like P(VB | NOUN), i.e., the probability of the current word having a tag of Verb given that the previous word had a NOUN tag.

The **emission probabilities** would be P(mary | NOUN) or P(will | VB), i.e., what is probability that given the word is, say, Mary given that the tag is a Noun.

Solving the problem using HMMs

We follow a generative tagging model to solve the problem using HMMs.
Let us assume a finite set of words V and a finite sequence of tags K. Then the set S will be the set of all sequences, tags pairs

$$< x1, x2, x3 \ldots xn, y1, y2, y3 \ldots yn >$$

such that,

$$n > 0 \ \forall \ x \in V \text{ and } \forall \ y \in K$$

The generative tagging model follows the following constraints:

1. For any $< x_1 \ldots x_n, y_1 \ldots y_n > \in S$, $p(x_1 \ldots x_n, y_1 \ldots y_n) \geq 0$
2. $\sum_{<x1 \ldots xn, y1 \ldots yn> \in S} p(x_1 \ldots x_n, y_1 \ldots y_n) = 1$

We need to obtain the following function from input to output:

$$f(x_1 \ldots x_n) = arg\ max_{y_1 \ldots y_n}\ p(x_1 \ldots x_n, y_1 \ldots y_n)$$

Thus, for a given sequence of words, the output is the tag sequence with the highest probability from the model. Now, we need to figure out the following:

1. How exactly do we define the generative model probability p(<x1, x2, x3 … xn, y1, y2, y3 … yn>)
2. How to estimate the parameters of the model
3. How to efficiently calculate

$$f(x_1 \ldots x_n) = arg\ max_{y_1 \ldots y_n}\ p(x_1 \ldots x_n, y_1 \ldots y_n)$$

Defining the Generative Model

We have used the Trigram Hidden Markov Model which can be defined using

- A finite set of states (here, tags)
- A sequence of observations (here, words encountered in a sentence)
- q(s|u, v) - probability of state “s” appearing right after observing “u” and “v” in the sequence of observations, i.e., Transition probability
- e(x|s) - probability of making an observation x given that the state was s

Thus the probability would be estimated as

$$p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) = \prod_{i=1}^{n+1} q(y_i | y_{i-2}, y_{i-1}) \prod_{i=1}^{n} e(x_i | y_i)$$

Estimating the model’s parameters

To determine the parameters, a training corpus is needed which would consist of several sentences with all words, already tagged correctly. We have used the **Brown Corpus of Present-Day American English** (widely known as Brown Corpus) as the training corpus. The parameters are estimated by reading various data off of the training corpus we have, and then deriving the maximum likelihood estimates as follows:

$$q(s|u,v) = \frac{c(u,v,s)}{c(u,v)}$$

$$e(x|s) = \frac{c(s \rightsquigarrow x)}{c(s)}$$

where,

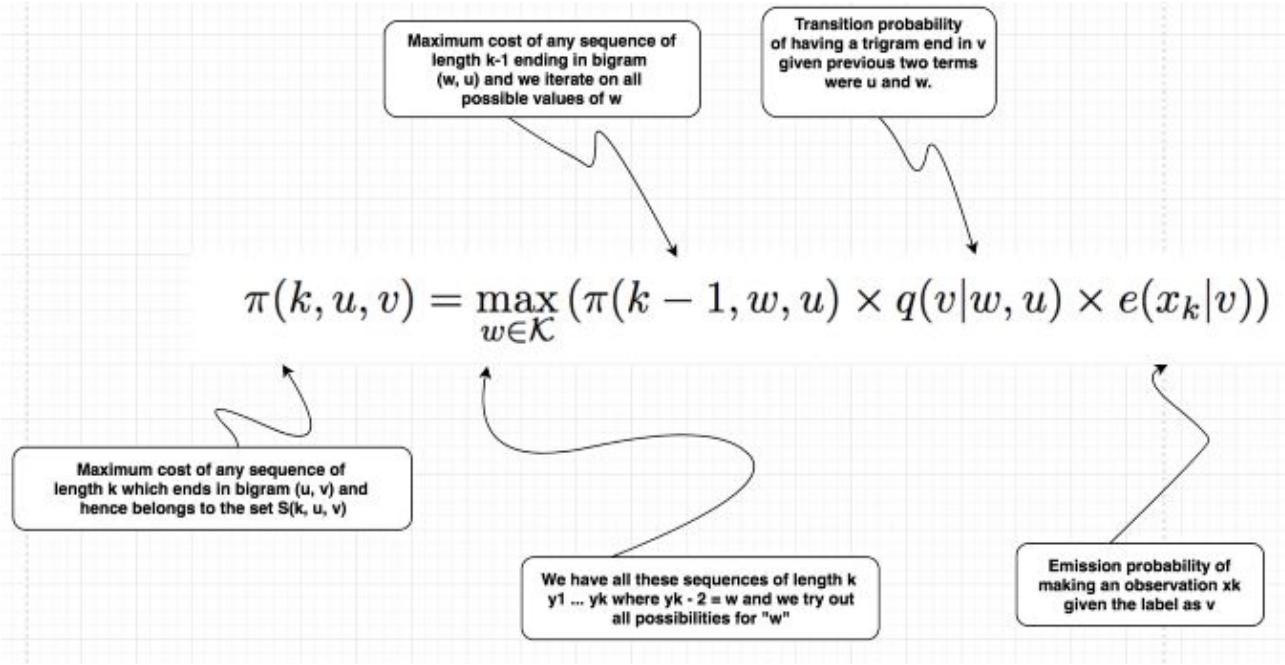
q(s|u, v) and *e(x|s)* mean the same as defined above
c(u, v, s) means the trigram count of tags u, v and s. It represents the number of times the three tags u, v and s occurred together in that order in Brown corpus.
c(u, v) is the bigram count of tags u and v in the order in the corpus.
c(s ⟶ x) is the number of times tag s and word x appear together.
c(s) is the past probability of a word being tagged as s.

Efficiently calculating $f(x_1 \ldots x_n)$ - Viterbi Algorithm

The end-goal here is to calculate value of $f(x_1 \ldots x_n)$ efficiently so that it does not become computationally expensive.
Following the brute-force approach, we have a large set of tags in the English grammar and thus, calculating probabilities for all paths generated by even a small sentence is computationally heavy. For example, taking 3 words in a sentence and 3 tags from English grammar result in $3^3 = 27$ paths. This number increases exponentially with increase in number of words per sentence and number of tags considered.

Therefore, we use the Viterbi algorithm, which is a DP-based algorithm.

The recursive step, which is the heart of the algorithm is as follows:



The definition is clearly recursive as we are trying to calculate one π term and we are using another one with a lower value of k in the recurrence relation for it.

$$\max_{y_1 \cdots y_{n+1}} p(x_1 \ldots x_n, y_1 \ldots y_{n+1}) = \max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n, u, v) \times q(STOP|u, v))$$

To use this algorithm, we attach two special start symbols “*” at the beginning of the sentence and one end symbol “STOP” is appended to the sentence. This is necessary as we are using the Trigram model, thus considering.

The pseudo-code for Viterbi algorithm which we have followed to write the code is:

Input: a sentence $x_1 \dots x_n$, parameters $q(s|u,v)$ and $e(x|s)$.
Initialization: Set $\pi(0,*,*) = 1$, and $\pi(0,u,v) = 0$ for all (u,v) such that $u \neq *$ or $v \neq *$.
Algorithm:

- For $k = 1 \dots n$,
  - For $u \in \mathcal{K}, v \in \mathcal{K}$,

$$\pi(k,u,v) = \max_{w \in \mathcal{K}} (\pi(k-1,w,u) \times q(v|w,u) \times e(x_k|v))$$

- **Return** $\max_{u \in \mathcal{K}, v \in \mathcal{K}} (\pi(n,u,v) \times q(\text{STOP}|u,v))$

The running time for the algorithm is $O(n|K|^3)$, hence it is linear in the length of the sequence, and cubic in the number of tags.

Python Code Explanation

- Requirements for executing the code:
  - Python - 3.8.3
  - Libraries -
    - numpy - 1.19.3
    - nltk - 3.5
    - time - 3.8.3
    - os - 3.8.3
    - pickle - 4.0
    - re - 2020.7.14
- Global parameters
  - STOP - special stop symbol to depict end of sentence
  - START - special start symbol, applied twice at the beginning of each sentence to facilitate trigram probability calculation
  - RARE - special symbol to mark words which are “rare”, i.e., which have frequency less than or equal to RARE_MAX_FREQ in the corpus
  - RARE_MAX_FREQ - maximum frequency of a word to be marked as RARE

    - LOG_OF_ZERO - when probability is zero, its logarithm is undefined, instead we use large negative number to denote log(0). This symbol is necessary as we are using logarithmic probability
    - PATH - store current directory path to facilitate writing to/from files
    - DEL - delimiter
    - DATA_PATH - directory path to locate input file
    - OUTPUT_PATH - directory path to locate output file
    - PARAMETERS_PATH - directory path to locate directory storing pickle files containing q and e values

- splitWordTags(sentences)
    - Receives a list of tagged sentences and processes each sentence to generate a list of words and a list of tags.
    - Start and stop symbols are included in returned lists, as defined by the constants START and STOP respectively.
    - Parameters
        - sentences: list of list of strings. Each sentence is a string of space-separated “WORD/TAG” tokens, with a newline character in the end
    - Returns
        - words, tags: 2d lists of string. Lists where every element is a list of the words/tas of a particular sentence

- QCalc(tags)
    - Takes tags from the training corpus and calculates tag trigram probabilities.
    - Parameters
        - tags: list of list of str. Each element of tags list is list of tags of particular sentence.
    - Returns
        - Qvalue: dictionary of tuple containing float values. The keys are tuples of str that represent the tag trigram. The values are the float log probability of that trigram.

- calculateKnown(words)
    - Takes the words from the training corpus and returns a set of all of the words that occur more than RARE_MAX_FREQ
    - Parameters

        - words: list of list of str. Each element of sentence_words is a list with str words of a particular sentence enclosed with START and STOP symbols.
    - Returns
        - knownWords: set of known words.

- replaceRare(sentences, knownWords)
    - Replaces rare words (words with frequency not greater than RARE_MAX_FREQ) with RARE symbol.
    - Parameters
        - sentences: list of list of string. Each sentence is a string of space-separated "WORD/TAG" tokens, with a newline character in the end.
        - knownWords: set of known words
    - Returns
        - replaced: list of list of string. List with the same structure as sentences but with words that is not in known_words replaced by RARE.

- Ecalc(toks, tags)
    - Takes tokens and tags from training corpus and calculates emission log probability of the word given the tag.
    - Parameters
        - toks: list of list of string. Each sentence is a string of space-separated "WORD/TAG" tokens, with a newline character in the end.
        - tags: list of list of string. Each element of tags list is list of tags of particular sentence.
    - Returns
        - eValue: log emission probabilities
        - tagList: set of all possible tags for this data set

- tagVITERBI(tokens, tagset, knownWords, Q, E)
    - Viterbi algorithm implementation
    - Parameters
        - tokens: list of string
        - tagset: set of string
        - knownWords: list of list of string. List of known words, those that occur more than RARE_MAX_FREQ times in training corpus
        - Q: dictionary of tuple with float values. Transition probabilities

        - E: dictionary of tuple with float values. Emission probabilities
    - Returns
        - tagged: list of tuples of string. List of sentence tokens together with its tags in tuples.

- outputQ
    - This function takes output from QCalc() and outputs it in the proper format

- outputE
    - This function takes output from Ecalc() and outputs it in the proper format

- outputTagged
    - Write the final output to the mentioned file name

- loadData
    - Load data from passed file name, read the data and return the file's contents

- saveObject
    - Convert files to pickle format and store them as dumps

- loadVITERBIPararms
    - Convert the pickle files to txt files for reading and using in Viterbi algorithm
    - Returns
        - Tuple of objects containing tagset, knownWords, Qvalue and Evalue

- main
    - Main function to incorporate all the functions
    - First extracts the tags and token from already present Brown corpus file
    - Then, transition probabilities (Qvalue) are calculated and stored in pickle files
    - Words with rare occurrence are given zero probability and replaced with RARE symbol in the words list
    - Emission probabilities (Evalue) are calculated from the filtered words and tags extracted and stored in pickle files
    - Individual words are extracted from input file and Viterbi algorithm is run on them
    - The final output is written to a file
    - Time taken to complete the tagging is also displayed.