Summary

Introduction to Syntactic Processing and PoS Tagging

You will be introduced to the following topics in this session:

- Syntax and syntactic processing
- Parts of speech and PoS tagging
- PoS tagging techniques: Rule-based and Hidden Markov Model
- PoS tagging code demonstration

Syntax and Syntactic Processing

Welcome to the first segment of this module on syntactic processing. We all have learnt in our schools how a particular language is built based on the grammatical rules.

Let’s take a look at the sentences given below.

‘Is EdTech upGrad company an.’

‘EdTech is an company upGrad.’

‘upGrad is an EdTech company.’

All of these sentences have the same set of words, but only the third one is syntactically or grammatically correct and comprehensible.

One of the most important things that you need to understand here is that in lexical processing, all of the three sentences provided above are similar to analyse because all of them contain exactly the same tokens, and when you perform lexical processing steps such as stop words removal, stemming, lemmatization and TFIDF or countvectorizer creation, you get the same result for all of the three sentences. The basic lexical processing techniques would not be able to identify the difference between the three sentences. Therefore, more sophisticated syntactic processing techniques are required to understand the relationship between individual words in a sentence.

Arrangement of words in a sentence plays a crucial role in better understanding the meaning of the sentence. These arrangements are governed by a set of rules that we refer to as ‘syntax’, and the process by which a machine understands the syntax is referred to as syntactic processing.

Syntax: A set of rules that govern the arrangement of words and phrases to form a meaningful and well-formed sentence

Syntactic processing: A subset of NLP that deals with the syntax of the language

You also looked at some examples such as chatbots, speech recognition systems and grammar checking software that use syntactic processing to analyse and understand the meaning of the text.

Now, let’s consider the following example.

‘In Bangalore, it is better to have your own transport.’

‘We use car pooling to transport ourselves from home to office and back.’

What do you think about the word ‘transport’ in these two sentences?

Parts of Speech

Let’s start with the first level of syntactic analysis: **PoS (Parts of Speech) tagging**. To understand PoS tagging, you first need to understand what parts of speech are.

Let’s consider a simple example given below.

‘You are learning NLP at upGrad.’

From your knowledge of the English language, you are well aware that in this sentence, ‘NLP’ and ‘upGrad’ are nouns, the word ‘learning’ is a verb and ‘You’ is a pronoun. These are called the parts of speech tags of the respective words in the sentence. A word can be tagged as a noun, verb, adjective, adverb, preposition, etc., depending upon its role in the sentence. These tags are called the PoS tags.

Assigning the correct tag, such as noun, verb and adjective, is one of the most fundamental tasks in syntactic analysis.

Suppose you ask your smart home device the following question.

‘Ok Google, where can I get the permit to work in Australia?’

The word 'permit' can potentially have two PoS tags: noun or a verb.

In the phrase 'I need a work permit', the correct tag of 'permit' is 'noun'.

On the other hand, in the phrase "Please permit me to take the exam", the word 'permit' is a 'verb'.

Parts of speech (PoS) are the groups or classes of words that have similar grammatical properties and play similar roles in a sentence. They are defined based on how they relate to the neighbouring words.

Assigning the correct PoS tags helps us better understand the intended meaning of a phrase or a sentence and is thus a crucial part of syntactic processing. In fact, all the subsequent parsing techniques (constituency parsing, dependency parsing, etc.) use the part-of-speech tags to parse the sentence.

Now, let’s understand the types of PoS tags available in the English language.

A PoS tag can be classified in two ways: open class and closed class.
Open class refers to tags that are evolving over time and where new words are being added for the PoS tag.

- Open class
  - Noun
  - Verb
  - Adjective
  - Adverb
  - Interjection

Some useful examples of open class PoS tags are as follows:

- Name of the person
- Words that can be added or taken from another language such as words taken from the Sanskrit language such as ‘Yoga’ or ‘Karma’
- Words that are nouns but can be used as verbs such as ‘Google’
- Words that are formed by a combination of two words such as football, full moon and washing machine

Closed class refers to tags that are fixed and do not change with time.

- Closed class
  - Prepositions
  - Pronouns
  - Conjunctions
  - Articles
  - Determiners
  - Numerals

Some examples of closed class PoS tags are as follows:

- Articles: a, an, the
- Pronouns: you and I

You can take a look at the universal tagsets used by the SpaCy toolkit here.

Note: Spacy is an open-source library used for advanced natural language processing, similar to NLTK, which you have used in lexical processing.

In this module on syntactic processing, you are going to use the SpaCy library to perform syntactical analysis.

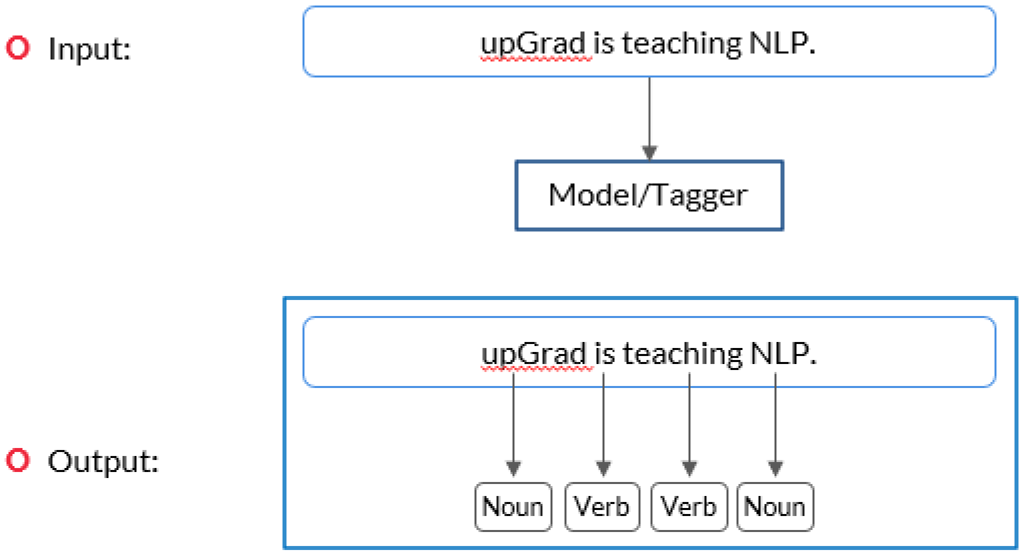
You do not need to remember all the PoS tags. You will pick up most of these tags as you work on the problems in the upcoming segments, but it is important to be aware of all the types of tags.

You can also refer to the alphabetical list of 36 part-of-speech tags used in the Penn Treebank Project, which is being used by the SpaCy library.

PoS Tagging

As of now, you have understood what the different part-of-speech tags are and their relevance in understanding the text. Now, let's understand how a machine assigns these PoS tags to the words in a sentence.

A PoS tagger is a model/algorithm that automatically assigns a PoS tag to each word of a sentence.



In this example, the input is "upGrad is teaching NLP.". When you put this sentence into a model or tagger, it will result in the output with respective PoS tags assigned to each word of the sentence such as 'upGrad' (noun), 'is' (verb), 'teaching' (verb) and 'NLP' (noun).

Let's take a look at another example given below.

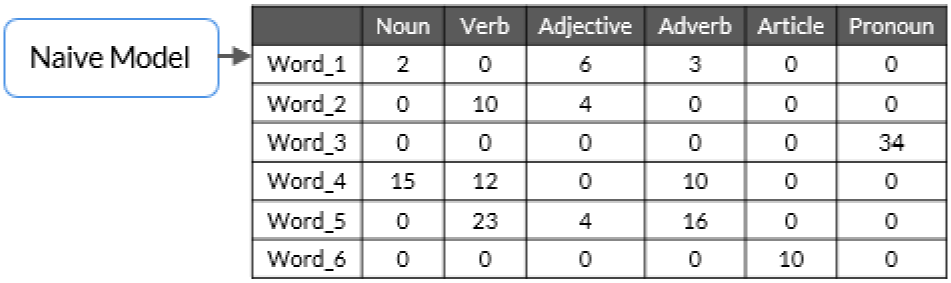
Tagger input: 'She sells seashells on the seashore.'

Tagger output: 'She(PRON) sells (VERB) seashells(NOUN) on(SCONJ) the(DET) seashore(NOUN).'

You can refer to the universal PoS tag list for finding tags corresponding to their parts of speech and also refer to the alphabetical list of part-of-speech tags used in the Penn Treebank Project, which is being used by the SpaCy library.

Now, let's try to understand how one can build a simple rule-based PoS tagger to assign PoS tags to individual words in a sentence.

Suppose you have been given a training dataset that comprises words and their respective PoS tags' count. This is visually demonstrated in tabular format below.



Naive Model

	Noun	Verb	Adjective	Adverb	Article	Pronoun
Word_1	2	0	6	3	0	0
Word_2	0	10	4	0	0	0
Word_3	0	0	0	0	0	34
Word_4	15	12	0	10	0	0
Word_5	0	23	4	16	0	0
Word_6	0	0	0	0	10	0

In this table, the word 'Word_1' occurs as a noun two times, and as an adjective, it occurs six times and so on in the training dataset.

The identification of PoS tags in the training dataset is done manually to predict the PoS tags of the test data.

In the table provided above, 'Word_1' appears as a noun two times, and as an adjective, it appears three times and so on. Now, suppose, you are given the following sentence (S).

S: "Word_4 Word_1 Word_3."

What will be the PoS tags of each word in this sentence?

you got the answer to the previous exercise, i.e., the PoS tags of the words of the sentence S will be as follows:

Word_4: Noun
Word_1: Adjective
Word_3: Pronoun

You assign the most frequent PoS tags that appear in the training data to the test dataset, and you realise that rule based tagger gives good results most of the time.

But, sometimes, it does not give satisfactory results because it does not incorporate the **context** of the word.

Let's take the following example.

Consider the word 'race' in both the sentences given below:

1. ‘The tortoise won the race.’
2. ‘Due to the injury, the horse will not be able to race today.’

In the first sentence, the word ‘race’ is used as a noun, but, in the second sentence, it is used as a verb. However, using the simple frequency-based PoS tagger, you cannot distinguish its PoS tags in different situations.

PoS Tagging: Hidden Markov Model - I

Let’s consider the following example; what does your mind process when you see the blank space at the end of this sentence?

Rahul is driving to _____.

Don’t you think that the blank space should be the name of a place?

How do you manage to identify that the blank space would be a name of the place?

Try to analyse your thoughts after reading this statement, when you see the word ‘Rahul’ who is driving to some place and hence, you reach to a conclusion that the blank space should be the name of a place (noun).

This means that you have sequentially looked at the entire sentence and concluded that the blank space should be the name of a place.

Sequence labelling is the task of assigning the respective PoS tags of the words in the sentence using the PoS tag of the previous word in the sentence.

Hidden Markov Model can be used to do sequence labelling, which means that it takes input of words in a sequence and assigns the PoS tags to each word based on the PoS tag of the previous word.

In the example ‘The tortoise won the race’, to decide the PoS tag of ‘race’, the model uses the PoS tag of the previous word, i.e., ‘the’, which is an article.

Now, let’s take a look at the following points and try to understand why Hidden Markov Model is called ‘Hidden’:

- When you observe (read/listen) a sentence, you only observe the words in the sentence and not their PoS tags; thus, PoS tags are hidden.
- You must infer these hidden tags from your observations, and that's why the Hidden Markov Model is called Hidden.

There are majorly two assumptions that HMM follows, which are as follows:

- The PoS tag of the next word is dependent only on the PoS tag of the current word.
- The probability of the next word depends on the PoS tag of the next word.

Before learning about HMM, you need to understand the two most important types of matrices, i.e., emission and transition matrices.

To build any machine learning model, you first need to train that model using some training data, and then, you need to use that model to predict the output on the test data.

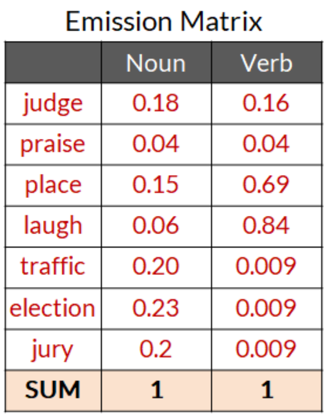
Here, the train data is the corpus of sentences, and you need to perform some manual tasks to assign the PoS tags to each word in the corpus. Once you manually assign the PoS tags to each word in the training corpus, you create two important matrices using the training dataset, which are as follows:

1. Emission matrix
2. Transition matrix

Note: We have used only two PoS tags, i.e., noun and verb, as of now to understand the concept in a simpler manner.

Emission matrix: This matrix contains all words of the corpus as row labels; the PoS tag is a column header, and the values are the conditional probability values.

Note: Conditional probability is defined as the probability of occurrence of one event given that some other event has already happened. You will get more idea about this in the following example.



Emission Matrix

	Noun	Verb
judge	0.18	0.16
praise	0.04	0.04
place	0.15	0.69
laugh	0.06	0.84
traffic	0.20	0.009
election	0.23	0.009
jury	0.2	0.009
SUM	**1**	**1**

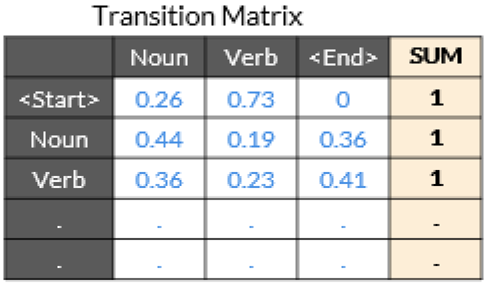
Next Whenever a noun appears in the training corpus, there is an 18% chance that it will be the word ‘judge’. Similarly, whenever a verb appears in the training corpus, there is an 84% chance that it will be the word ‘laugh’.

So, here, 0.18 is the probability of occurrence of the word ‘judge’ given that there will be a noun at that place. In a similar way, 0.16 is the probability of occurrence of the word ‘judge’ given that there will be a verb at that place.

Transition matrix: This matrix contains PoS tags in the column and row headers. Let’s try to understand the conditional probability values that have been given in the following table.

Let’s take a look at the first row of the table; it represents that 0.26 is the probability of occurrence of a noun at the start of the sentence in the training dataset. In the same way, 0.73 is the probability of occurrence of a verb at the start of the sentence in the training dataset.

If you take a look at the second row, then you will see that 0.44 is the probability of occurrence of aoun just after a noun in the training dataset, and similarly, 0.19 is the probability of occurrence of a verb just after a noun in the training dataset.



Transition Matrix

	Noun	Verb	<End>	SUM
<Start>	0.26	0.73	0	**1**
Noun	0.44	0.19	0.36	**1**
Verb	0.36	0.23	0.41	**1**
-	-	-	-	-
-	-	-	-	-

Essentially, the transition matrix gives the information of the next PoS tag considering the current PoS tag of the word.

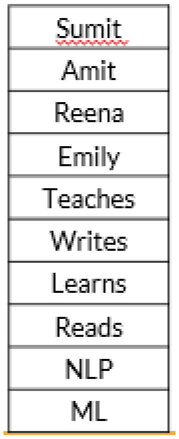
Hidden Markov Model - II

In HMM, you understood that **transition and emission matrices** specify the probabilities of transition between tags and the emission of the word from the tag, respectively.

In this segment, you will learn how, with the use of transition and emission matrices, you can assign the correct PoS tag sequence based on the probability values.

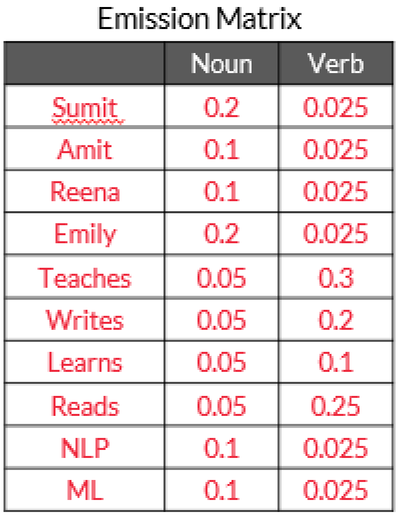
Please note that you will not be gaining an in-depth understanding of HMM in this module, as it is not industrially relevant at all. You will only gain an intuitive understanding of how HMM works on sequence labelling of PoS tags.

Suppose you have the following corpus of the training dataset.



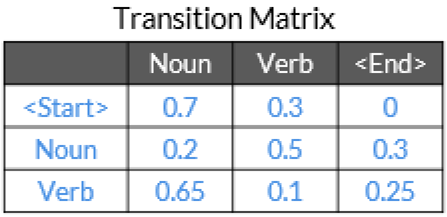
Sumit
Amit
Reena
Emily
Teaches
Writes
Learns
Reads
NLP
ML

The emission and transition matrices of this corpus are as follows:



Emission Matrix

	Noun	Verb
Sumit	0.2	0.025
Amit	0.1	0.025
Reena	0.1	0.025
Emily	0.2	0.025
Teaches	0.05	0.3
Writes	0.05	0.2
Learns	0.05	0.1
Reads	0.05	0.25
NLP	0.1	0.025
ML	0.1	0.025



Transition Matrix

	Noun	Verb	<End>
<Start>	0.7	0.3	0
Noun	0.2	0.5	0.3
Verb	0.65	0.1	0.25

Suppose you have been given the following test sentence to predict the correct PoS tags of the words.

S: “Sumit teaches NLP.”

As of now, we are only considering the two PoS tags, i.e., noun (N) and verb (V).

There are many combinations of PoS tags possible for the sentence ‘S’ such as NVN, NNN, VVV and VVN or NNV.

If you calculate the total number of combinations for this example, then you will see that there are only noun and verb tags; hence, 2^3 will be the total number of combinations possible.

Let’s consider the two sequences as of now, which are NNN and NVN.

Calculate the score for the NNN sequence.

Score of NNN:
[P(Start-Noun) * P(Sumit|Noun)] * [P(Noun-Noun)*P(teaches|Noun)] * [P(Noun-Noun) * P(NLP|Noun)]

(0.7*0.2) * (0.2*0.05) * (0.2*0.1) = 0.000028

Score of NVN:
[P(Start-Noun)*P(Sumit|Noun)] * [P(Noun-Verb)*P(teaches|Verb)] * [P(Verb-Noun) * P(NLP|Noun)]

(0.7*0.2) * (0.5*0.3) * (0.65*0.1) = 0.001365

You get the maximum score for the NVN sequence; hence the model assigns the noun to ‘Sumit’, the verb to ‘teaches’ and another noun to ‘NLP’ in the test sentence.

In practice, you do not have to calculate these scores from scratch. You will be using an already prepared toolkit to implement HMM, but it is important to be aware of the basic techniques used in PoS tagging.

Although you are not going to implement these PoS tagging techniques in the real-life scenario, you are going to use the SpaCy library to tag the correct PoS tags that are based on the neural network models. It is important to have an intuitive understanding of these techniques, including the rule-based tagger and HMM, to understand how a PoS tagger works.

With this, you have understood the theory part of PoS tagging.

PoS Tagging Application Part- I

You learnt about the underlying process of PoS tagging, i.e Hidden Markov Model(HMM). In this segment, you have gone through a coding exercise to see how PoS tagging can be used to solve the problem.

We are not going to design and implement the rule-based tagger or a sequential tagger like HMM in the real world. However, it is good to have an intuitive understanding of how a PoS tagger can be built.

For the implementation of PoS tagging, you are going to use a Python library, i.e., Spacy.

Please note that it is not important to deep dive into the working of the Spacy library. The working of Spacy is based on neural networks which is out of the scope of this course. Spacy works well in different applications of NLP such as PoS tagging, parsing and lexical processing steps. You just need to learn how to use this library for various applications.

Now, let’s learn how one can use PoS tagging in applications like Heteronyms identification using the Spacy library.

Consider the following two sentences and pay attention to the pronunciation of word **‘wind’** while you read:

- The doctor started to wind the bandage around my finger.
- The strong wind knocked down the tree.

You can listen to these sentences in the Google Translator application.

As you might have noticed, the pronunciation of **‘wind’** is different in both the sentences, though its spelling is the same. Such words are known as **Heteronyms.**

Heteronyms are words that have the same spelling but mean differently when pronounced differently.

But how do machines like Alexa and Google Translator detect these heteronyms?

Let’s see the following sentence:

‘She wished she could desert him in the desert.’

Google translator pronounces ‘desert’ differently based on its PoS tag. You can see that both the instances of the word ‘desert’ have different PoS tags, one is a verb and other one is a noun.

PoS Tagging Application Part - II

Let’s try to understand each line of code one by one.

```
model = spacy.load(“en_core_web_sm”)
```

- **‘en’** stands for English language, which means you are working specifically on English language using the Spacy library.
- **‘core’** stands for core NLP tasks such as lemmatization or PoS tagging, which means you are loading the pre-built models which can perform some of the core NLP-related tasks.
- **‘web’** is the pre-built model of the Spacy library which you will use for NLP tasks that are trained from web source content such as blogs, social media and comments.
- **‘sm’** means small models which are faster and use smaller pipelines but are comparatively less accurate. As a complement to ‘sm’, you can use **‘lg’** or **‘md’** for larger pipelines which will be more accurate than ‘sm’.

Visit this link to know more about the Spacy package.

So, you have loaded the ‘en_core_web_sm’ model into the object ‘model’. Now, let’s apply the model on the input sentence, i.e., ‘She wished she could desert him in the desert’, using the following line of code.

```
#Use the model to process the input sentence
tokens = model("She wished she could desert him in the desert.")
```

So, you get the words and its part of speech and tags in the variable ‘tokens’. Now, to print the tokens, part of speech and PoS tags, you need to apply a ‘for’ loop on this ‘tokens’ object as follows:

```
# Print the tokens and their respective PoS tags.
for token in tokens:
    print(token.text, "--", token.pos_, "--", token.tag_)
```

So, when you print ‘token.pos_’, it prints the part of speech and using ‘token.tag_’, it prints the PoS tags corresponding to each token in a given sentence.

Please note that the ‘token.pos_’ gives you the part of speech which is defined in Spacy’s universal part of speech tags that you can get from this link. Moreover, to give the PoS tags using ‘token.tag_’, Spacy uses the PoS tags provided by the Penn treebank that you can get from this link.

When you get the PoS tags of each token of the sentence ‘She wished she could desert him in the desert’, then you will observe that the PoS tag of the word ‘desert’ is different in both of the instances. At one place, it is working as a **verb** and in another instance, it is working as a **noun.**

Word sense disambiguation (WSD): WSD is an open problem in computational linguistics concerned with **which sense of word** is used in a sentence. It is very difficult to fully solve the WSD problem. Google, however, has partially solved it.

Let’s try to listen to the pronunciation of the word ‘bass’ in the following sentence in Google Translator.

‘The bass swam around the bass drum on the ocean floor’.

However, as you have seen, the use of PoS tagging in heteronyms detection can be one of the prominent solutions to remove ambiguity in the sentence.

Let’s take the example of the following sentence:

‘She wished she could desert (verb) him in the desert (noun)’.

Here, the word ‘desert’ has two PoS tags based on its uses. At one place, it is working as a verb and at another place, it is working as a noun.

But what if the heteronyms have the same PoS tag but different pronunciation?

PoS tagging works pretty accurately to detect heteronyms but fails to distinguish between the words having the same spelling and the same PoS tags. Let’s consider the following example:

‘The bass swam around the bass drum on the ocean floor’.

When you implement the model on this sentence, you get the list of PoS tags of each token in the sentence. As you noticed that the word ‘bass’ is working as a noun at both of the places, but it should have different pronunciation at both the instances.

So, the problem when the system is not able to identify the correct pronunciation of the words which have the same PoS tag but different meanings in different contexts can be considered under the WSD problem.

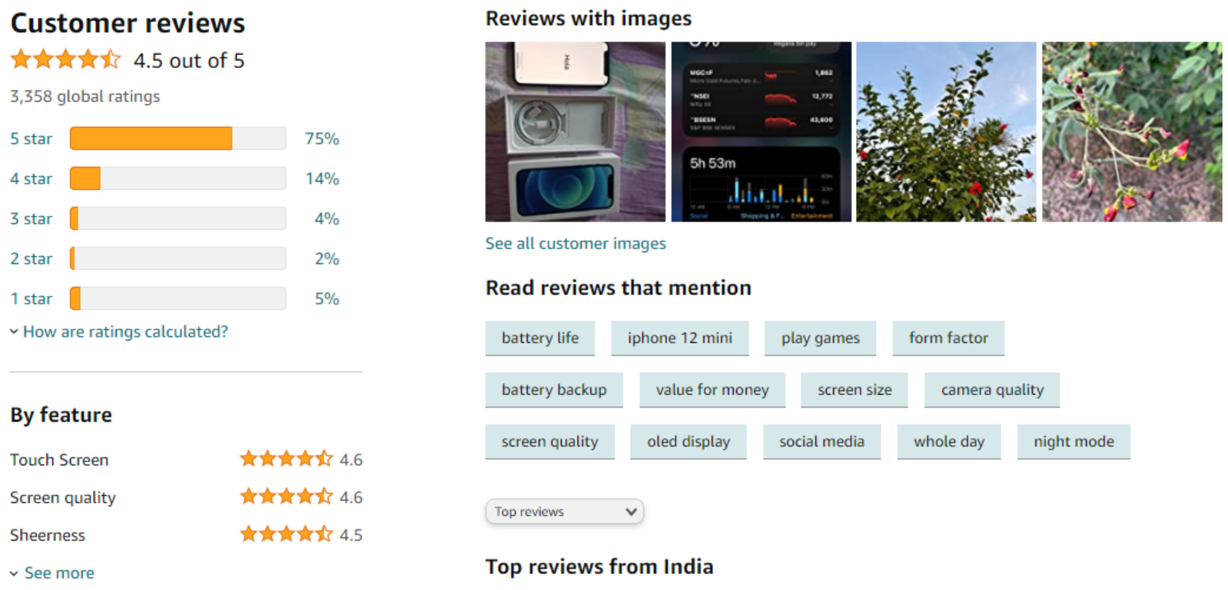
Please note that this is just one of the dimensions of WSD. WSD is altogether a broader area to discover and it is an open problem in computational linguistics concerned with identifying which sense of a word is used in a sentence.

PoS Tagging Case Study Part - I

Now, let’s come to a very interesting case study using PoS tagging. Here, we will provide you with the problem statement and some hints to produce output of each step involved to solve the case study.

Let’s understand the problem statement first.

You have used this feature of Amazon many times before buying any product from the website where you look into the reviews of the product category wise. Let’s look at the following.



Customer reviews

4.5 out of 5

3,358 global ratings

5 star	75%
4 star	14%
3 star	4%
2 star	2%
1 star	5%

How are ratings calculated?

By feature

Touch Screen	4.6
Screen quality	4.6
Sheerness	4.5

See more

Reviews with images

See all customer images

Read reviews that mention

battery life | iphone 12 mini | play games | form factor | battery backup | value for money | screen size | camera quality | screen quality | oled display | social media | whole day | night mode

Top reviews from India

The image is the Amazon page of Apple iPhone 12 Mini mobile phone. As you can see, there is categorisation of reviews into features based on the reviews’ text submitted by the users. These features can be ‘battery life’, ‘value for money’, ‘screen size’ and so on.

It is interesting to know that these features have been created to categorise the reviews after looking into the reviews' text given by multiple users. So, when you click on let's say 'battery life', you will get all the reviews related to battery life of this phone.

Please note that in this case study, we are not going to segregate or categorise the reviews, but we are going to find out the top product features that people are talking about in their reviews. The categorisation of reviews among the product features is just a task to assign each review to its category which can be done using simple coding steps once the main NLP steps are done.

Please note that as this use case is not a graded component and is designed to be solved as a case study, you have been provided with the data set and the well-commented codes to solve it. ***It is not at all mandatory to solve each of the problems; you can watch the videos directly.***

Let's first find out the PoS tags of each token in the following sentences and observe the PoS tags of the words 'screen', 'battery' and 'speaker'.

sent1 = "I loved the screen on this phone".
sent2 = "The battery life on this phone is great".
sent3 = "The speakers are pathetic".

You have an understanding of getting the PoS tags from the previous segments and can write the code to get the PoS tag of each token for these three sentences. Now, based on the output that you get after having the PoS tags,

You observed that the product features such as screen, battery and speaker have a PoS tag of a Noun.

An important thing to note here is that suppose we are able to find the frequency count of all the nouns in our data set, then by looking at top-n nouns, we can find out what product features people are talking about.

You have been given a Samsung phone reviews data set with file name 'Samsung.txt' file. It contains the reviews about the phone. This data set is a txt file, and each review is in a new line. In the notebook, you have been given the code to load the data set into the notebook.

In this task, you need to calculate the total number of reviews in the data set where each review is in a new line. Let's solve the next question based on the defined task.

In the next part, you need to check whether the hypothesis, i.e., the product features that we are trying to extract are acting as nouns in the real-life data set such as 'Samsung.txt'. But before that, let's first try to understand why lemmatization of words is needed before extracting the nouns from the data set.

In this task, you need to identify the right approach before extracting the nouns from the data set.

So, you can see that there are many nouns in the review text but all of them are not at all relevant to identify the features. So, what should be the approach to get relevant nouns from the review text?

Let’s follow the approach that first calculates the PoS tags of each token and then converts them into their lemma form so that each word turns into its root form. Now, once you find the lemma form of each token, you need to count the frequency of each word which is acting as a noun.

You need to calculate the frequency of each noun after performing the lemmatization in the first review of the data set and arrange them in the descending order of their frequency of occurrence as a noun.

So, as you can see, when you restrict yourself to the top occurring noun, you can get the desired result. Now, let’s calculate the frequency of occurrence of nouns for each token in the first 1,000 reviews of the data set.

Let’s extract all the nouns from the reviews (first 1,000 reviews) and look at the top five most frequently lemmatised noun forms.

You learnt that the top frequently occurring noun PoS tags can work as product features.

Gunnvant has used the **‘tqdm’** library to track the time of the processing of 1,000 reviews. You have seen that to process the first 1,000 reviews, it takes 17 seconds and, hence, to process the entire data set, it will take around 782 seconds which is approximately 13 minutes.

PoS Tagging Case Study Part - II

You need to extract all the nouns from all of the reviews and look at the top 10 most frequently lemmatised noun forms. You are already aware that it will take too much time to process the entire data set (approximately 13 minutes that we calculated in the previous segment). To reduce the processing time, you can use the following line of code when you load the Spacy model.

```
# shorten the pipeline loading
nlp=spacy.load('en_core_web_sm',disable=['parser','ner'])
```

However, for solving the next question, please use the aforementioned lines of code.

Gunnvant has used the following lines of code to reduce the processing time:

```
# shorten the pipeline loading
nlp=spacy.load('en_core_web_sm',disable=['parser','ner'])
```

When you load the Spacy using ‘en-core_web_sm’, the system finds everything like PoS tags, parsing dependencies, pre-built NER, lemmatised forms and so on. When you disable the ‘parser’ and ‘ner’, you are actually telling the system to not perform these tasks which eventually reduces the time to process the data set. As you can see, the time to process the entire data set was calculated as approximately 13 minutes earlier. However, in this case, the total time to process the entire data set is 4.6 minutes only.

Let’s see some pointers

- Now, we know that people mention battery, product, screen, etc. However, we still do not know in what context they mention these keywords.
- The most frequently used lemmatized forms of nouns inform us about the product features people are talking about in product reviews.
- In order to process the review data faster, Spacy allows us to use the idea of enabling parts of the model inference pipeline via the spacy.loads() command and the disable parameter.

Now, let’s come to a very interesting exercise. You have been able to identify the keywords or features such as ‘battery’, ‘screen’ and ‘phone’. Now, let’s try to see in what context these keywords occur using the following examples:

- Good battery life
- Big screen size
- Large screen protector
- Long battery lasts

In these examples, you can see that the keyword ‘battery’ occurs in the context of ‘good battery life’ or ‘screen’ occurs in the context of ‘big screen size’ and so on.

You have been given a sentence and need to identify the prefix and the suffix of the keyword ‘battery’. Using prefixes and suffixes, you can identify the context of keywords.

Now, let’s extract all suffixes and prefixes of the keyword ‘battery’ in the whole ‘Samsung.txt’ data set.

```
#define the regular expression pattern.
pattern = re.compile("\w+\sbattery\s\w+")

#apply the regular expression pattern for the keyword ‘battery’ to all the reviews of the
data set using ‘findall’ function.
prefixes_suffixes = re.findall(pattern,samsung_reviews)
# Add prefixes and suffixes in separate arrays for the entire
dataset corresponding to the keyword ‘battery’.
```

```
prefixes = []
suffixes = []
for p in prefixes_suffixes:
    l = p.split(" ")
    prefixes.append(l[0].lower())
    suffixes.append(l[-1].lower())
```

# print the prefixes in the descending order of their count corresponding to the keyword ‘battery’.

```
pd.Series(prefixes).value_counts().head(5)
```

# print the suffixes in the descending order of their count corresponding to the keyword ‘battery’.

```
pd.Series(suffixes).value_counts().head(5)
```

A. You need to separate the suffixes and prefixes of all the combinations where a keyword ‘battery’ is present and then print the count of each prefix corresponding to the keyword ‘battery’ in a descending order.

As you have seen in the aforementioned example, there are many prefixes and suffixes which are less relevant to combine with keywords ‘battery’ such as ‘the battery’ and ‘that phone’ because these words are mainly stop words and do not carry any relevant information.

So, it will be better if you remove the stop words from the ‘prefixes’ and the ‘suffixes’.

You need to remove the stop words from the arrays of ‘prefixes’ and ‘suffixes’ for the keyword ‘battery’.

Before attempting the next question, you need to run the code that has been asked in the previous two questions so that you get the arrays named ‘prefixes’ and suffixes’ for the keyword ‘battery’.

So, you have learnt to get the most appropriate combination of keywords and its prefixes and suffixes after eliminating the stop words.

Summary

Parsing

You introduced to the following topics:

- Constituency parsing

- Dependency parsing
- Parsing code demonstration

Constituency Parsing

By now, you learnt about PoS tagging in detail, which is, however, not enough for understanding the complex grammatical structures and ambiguities in sentences that most human languages consist of.

A key task in syntactic processing is parsing. It means to break down a given sentence into its 'grammatical constituents'. Parsing is an important step in many applications that helps us better understand the linguistic structure of sentences.

You need to learn techniques that can help you understand the grammatical structures of complex sentences. **Constituency parsing** and **dependency parsing** can help you achieve that.

Let’s understand parsing through an example. Suppose you ask a question answering (QA) system, such as Amazon's Alexa or Apple's Siri, the following question.

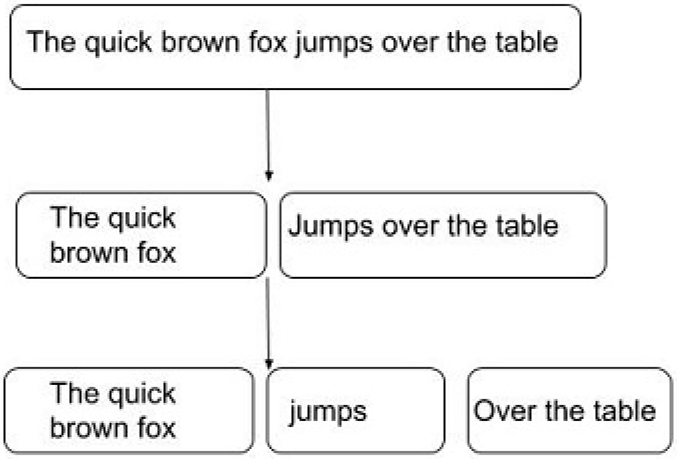
"Who won the FIFA World Cup in 2014?"

The QA system can respond meaningfully only if it understands that the phrase ‘FIFA World Cup' is related to the phrase 'in 2014'. The phrase 'in 2014' refers to a specific time frame and thus modifies the question significantly. Finding such dependencies or relations between the phrases of a sentence can be achieved using parsing techniques.

Let's take another example sentence to understand how a parsed sentence looks like.

"The quick brown fox jumps over the table."

The figure given below shows the three main constituents of this sentence.



This structure divides the sentence into the following three main constituents:

- 'The quick brown fox', which is a noun phrase
- 'jumps', which is a verb phrase
- 'over the table', which is a prepositional phrase

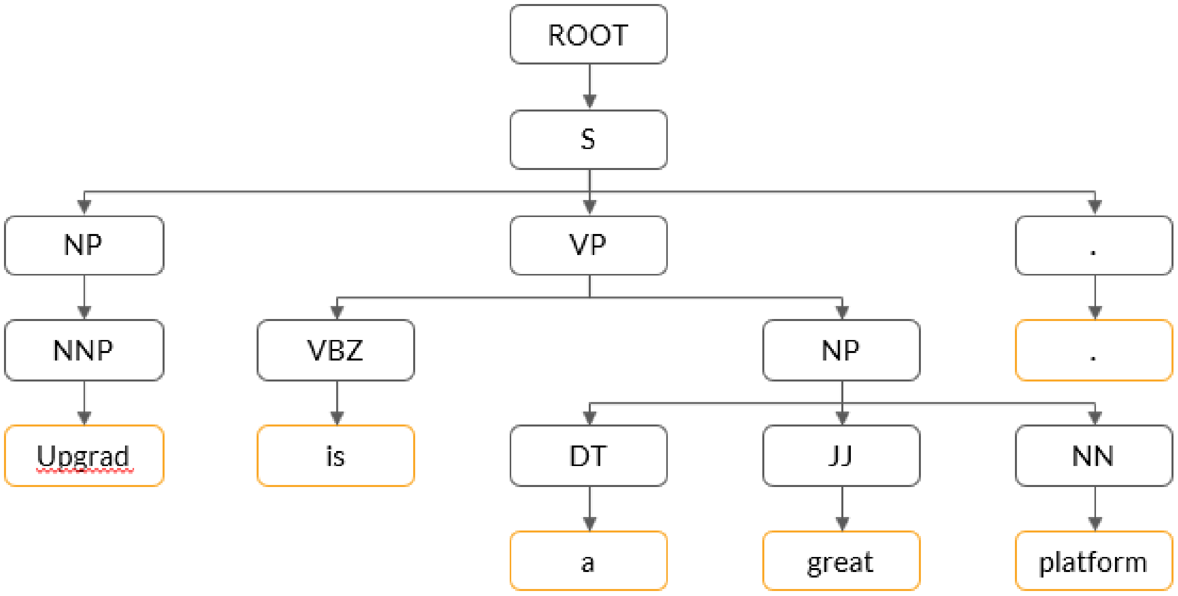
Constituency parsing is the process of identifying the constituents in a sentence and the relation between them.

For example, “upGrad is a great platform.”

Here, ‘upGrad’ is the noun phrase, and ‘is a great platform’ is the verb phrase.

The figure shown below represents the parse tree that shows how parsers implement parsing based on grammar.

CONSTITUENCY PARSE TREE



To summarise the chart given above, a constituent parse tree can be divided into three levels, which are as follows:

1. **Sentence constituent:**

S (upGrad is a great platform)

NP: Noun phrase (upGrad)

VP: Verb phrase (is a great platform)

2. **Sentence words:** 'upGrad', 'is', 'a', 'great', 'platform'
3. **Part-of-speech tags:** NNP, VBZ, NP, DT, JJ, NN

A constituency parse tree always contains the words of a sentence as its terminal nodes. Usually, each word has a parent node containing its part-of-speech tag (noun, adjective, verb, etc.).

All the other non-terminal nodes represent the constituents of the sentence and are usually one of verb phrases, noun phrases or prepositional phrases (PP).

In this example, at the first level below the root, the sentence has been split into a noun phrase, made up of the single word "upGrad" and a verb phrase "is a great platform". This means that grammar contains a rule such as S -> NP VP, meaning that a sentence can be created with the concatenation of a noun phrase and a verb phrase.

Similarly, the noun phrase is divided into a determiner, adjective and noun.

To summarise, constituency parsing creates trees containing a syntactical representation of a sentence according to a context-free grammar rule. This representation is highly hierarchical and divides the sentences into its single phrasal constituents.

Constituency parsing has been used in word processing software, grammar checking software, question answering system, etc..

You also looked at the example '**We saw the Statue of Liberty flying over New York**.' Although having the same arrangement of words, the sentence can be interpreted in the following two ways:

1. Person saw that the 'Statue of Liberty' was flying.
2. A person is flying over New York he/she saw 'Statue of Liberty' from the top.

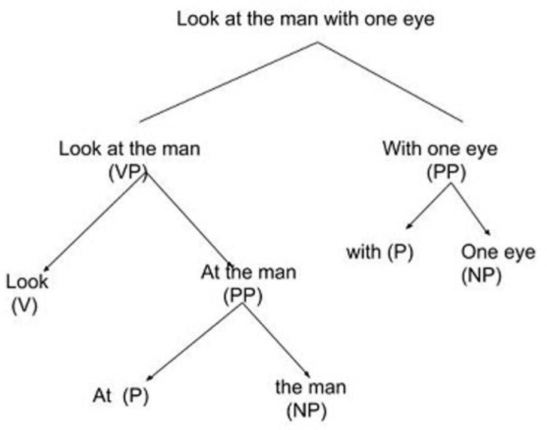
To understand ambiguity with a parse tree, let's consider the following sentence.

"Look at the man with one eye."

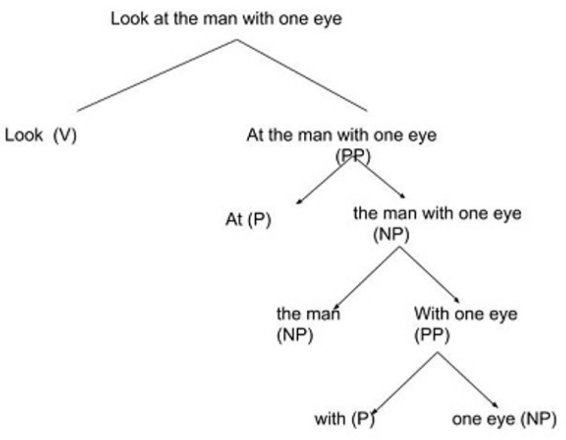
This sentence may have the following two meanings:

1. Look at the man using only one of his eyes.
2. Look at the man who has one eye.

Their respective parse trees are shown in the figure given below.



Look at the man using only one of your eyes



Look at the man who has one eye

There are two parse trees possible for this sentence; if we are able to identify the relationship among the words instead of looking at individual constituents or PoS tags, then understanding each word with other words will be easier, and the machine will be able to understand the syntax or meaning of the sentence better. This relationship structure can be drawn using the dependency parsing technique.

In general, since natural languages are inherently ambiguous (at least for computers to understand), there are often cases where multiple parse trees are possible, such as those in the example provided above. So, to understand the relationship between different words, you need to use **dependency parsing,** which you will learn in the next segment.

Note: You do not have to build these parse trees from scratch; there are various pre-trained models that you will use for your applications.

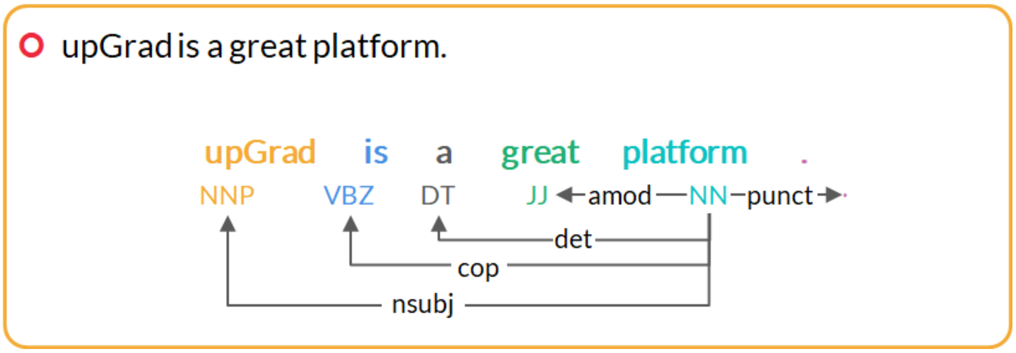
Dependency Parsing

Until now, we discussed constituency parsing where groups of words or constituencies form the basic structure of a parse tree. In this segment, we will introduce an alternate paradigm of grammar called **dependency grammar** and related dependency parsing techniques.

In dependency parsing, we do not form constituencies (such as noun phrase and verb phrase) but rather establish a dependency between the words themselves.

Dependency parsing identifies the relation between words and produces a concise representation of these dependencies.

In the example ‘upGrad is a great platform’, constituency parsing fails to explain how ‘great’ is related to the two nouns of the sentence ‘upGrad’ and ‘platform’. Is ‘great’ related more to ‘upGrad’ than to ‘platform’? Whereas, dependency parsing tells us that ‘great’ is a modifier of ‘platform’. Similarly for the two nouns, ‘upGrad’ is the subject of another noun ‘platform’.

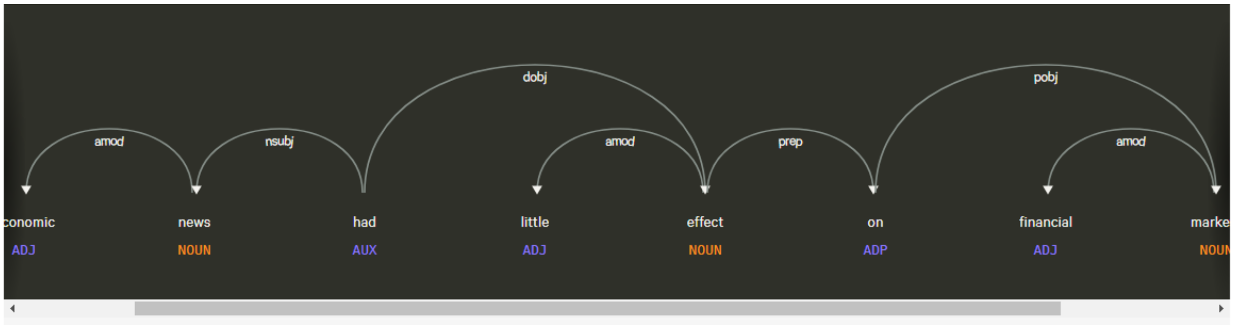


You can take a look at the universal dependency tags here that SpaCy uses.

Now, let’s take another example of the dependency parse tree. It is as follows:

Sentence: “Economic news had little effect on financial markets.”

You can visualise the dependency parse of this sentence here. Also, in the diagram shown below, we have merged the phrases such as 'Economic news' and 'little effect'.



Let’s identify the role of each word one by one, starting with the root verb.

The word **'had'** is the root.

The phrase ‘**Economic news’** is the nominal subject (nsubj).

The phrase '**little effect**' is the direct object (dobj) of the verb **'had'.**

The word **'on'** is a preposition associated with '**little effect**'.

The noun phrase '**financial markets**' is the object of **'on'.**

Now, let's take a look at the role of each word in the parse.

The word **'Economic'** is the modifier of **'news'**.

news -> amod -> economic

The words **'financial'** and **'little'** modify the words **'markets'** and **'effect',** respectively.

effect -> amod -> little
markets -> amod -> financial

The two words **'on'** and **'markets'** have no incoming arcs. The word **'on'** is dependent on the word **'effect'** as a nominal modifier.
effect -> prep -> on

The word 'markets' is an object of the word 'on'.
on -> pobj -> markets

Moving further, you need to keep the following points in mind while understanding the dependency parse model:

- Each word is a node in a parse tree.
- There is exactly one node with no incoming arc (root).
- Each non-root node has exactly one incoming arc.
- There is a unique path from the root node to each non-root node.

In this way, dependency parsers relate words to each other.

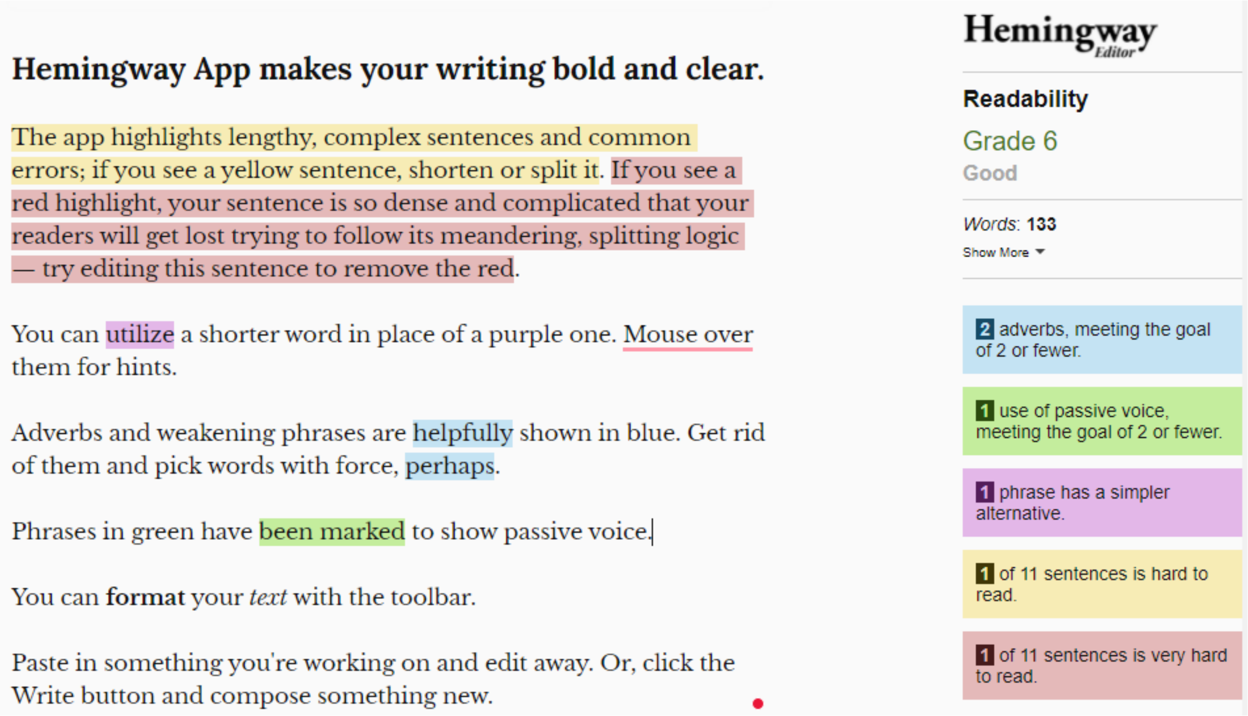
Note: You do not have to build these parsers from scratch; there are various pre-trained models that you will use for your applications. However, it is important to be aware of how these parsers work.

Parsing – Python Implementation Part - I

You learnt about one of the applications of parsing in which you can use parsing to identify the active and passive sentences.

Have you ever used the Grammarly or Hemingway applications to check grammar and spelling errors in your document or mail?

Let's look into the following snapshots of how these applications identify grammar and spelling errors.



Hemingway App makes your writing bold and clear.

The app highlights lengthy, complex sentences and common errors; if you see a yellow sentence, shorten or split it. If you see a red highlight, your sentence is so dense and complicated that your readers will get lost trying to follow its meandering, splitting logic — try editing this sentence to remove the red.

You can utilize a shorter word in place of a purple one. Mouse over them for hints.

Adverbs and weakening phrases are helpfully shown in blue. Get rid of them and pick words with force, perhaps.

Phrases in green have been marked to show passive voice.

You can **format** your *text* with the toolbar.

Paste in something you're working on and edit away. Or, click the Write button and compose something new.

Readability
Grade 6
Good

Words: 133

2 adverbs, meeting the goal of 2 or fewer.

1 use of passive voice, meeting the goal of 2 or fewer.

1 phrase has a simpler alternative.

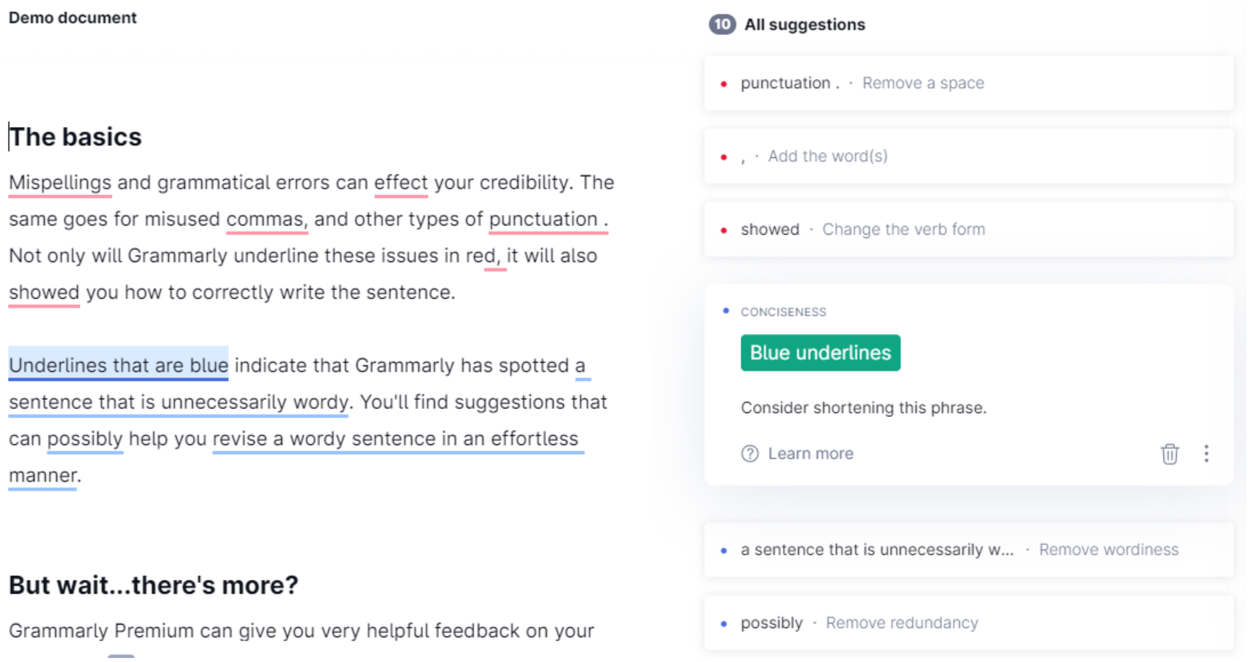
1 of 11 sentences is hard to read.

1 of 11 sentences is very hard to read.

Source: https://hemingwayapp.com/

In the image, the Hemingway application identifies the various instances where the document needs to be corrected grammatically. You can also see that it suggests a better version of a sentence if that particular sentence is difficult to read (highlighted in pink). The green highlights tell us that the sentence is in a passive voice.

The Grammarly application works in a similar manner. It is also a tool used to rectify grammar or spelling errors in a text corpus.



Demo document

The basics

Mispellings and grammatical errors can effect your credibility. The same goes for misused commas, and other types of punctuation . Not only will Grammarly underline these issues in red, it will also showed you how to correctly write the sentence.

Underlines that are blue indicate that Grammarly has spotted a sentence that is unnecessarily wordy. You'll find suggestions that can possibly help you revise a wordy sentence in an effortless manner.

But wait...there's more?

Grammarly Premium can give you very helpful feedback on your

10 All suggestions

- punctuation . · Remove a space
- , · Add the word(s)
- showed · Change the verb form
- CONCISENESS

  Blue underlines

  Consider shortening this phrase.

  Learn more
- a sentence that is unnecessarily w... · Remove wordiness
- possibly · Remove redundancy

Source: https://app.grammarly.com/ddocs/677685883

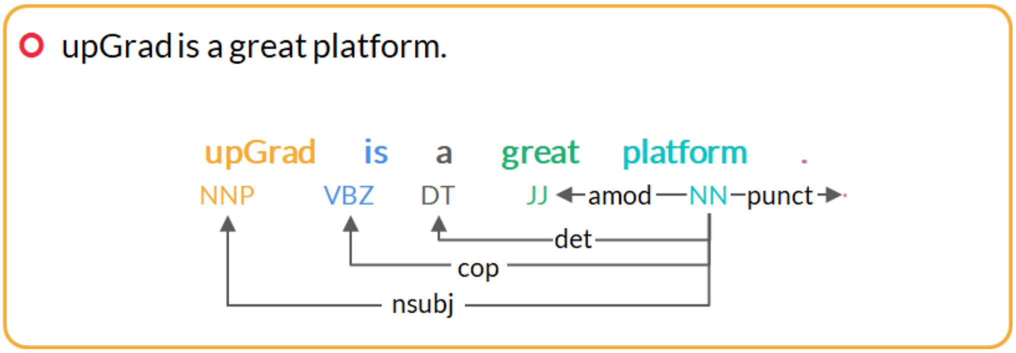
Now, it is always recommended to avoid using passive sentences in your text, document or email. So, here, you are going to design an application to identify whether a particular sentence is in active or passive voice.

Let’s try to understand the code one by one:

```
import spacy
from spacy import displacy
import pandas as pd
nlp = spacy.load("en_core_web_sm")
```

In the very first step, you imported the spaCy library. Please note, the spaCy is an open-source library for advanced natural language processing. It helps you build applications that process and understand the text.

You imported the ‘displacy’ library which is a modern syntactic dependency visualiser. You also saw the dependency parsing diagram in the previous segment for the following sentence.



- upGrad is a great platform.

You can visualise the dependency parse tree as shown earlier using the ‘displacy’ library.

After this, you loaded spacy.load("en_core_web_sm") in ‘nlp’. You can learn more about en_core_web_sm from this link.

After this, you defined some active and passive sentences as follows:

```
active = ['Hens lay eggs.',
         'Birds build nests.',
         'The batter hit the ball.',
         'The computer transmitted a copy of the manual']
passive = ['Eggs are laid by hens',
           'Nests are built by birds',
           'The ball was hit by the batter',
           'A copy of the manual was transmitted by the computer.']
```

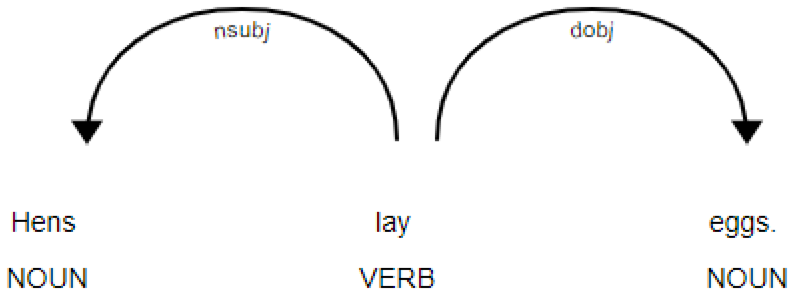
There are four active sentences and corresponding passive sentences. Now, let’s consider the very first sentence active[0] of the array ‘active’ and find its dependency tags using the following code lines.

```
doc = nlp(active[0])
for tok in doc:
    print(tok.text,tok.dep_)
```

So, you have parsed the ‘active[0]’ sentence through ‘en_core_web_sm’ and got the output as a list of tokens and their corresponding dependency tags.

You can build its dependency tree structure using the ‘displacy’ as follows:

```
displacy.render(doc, style="dep")
```



Please note that to run the aforementioned code in Google Colab, you need to add ***jupyter = True*** as follows:

```
displacy.render(doc, style="dep" , jupyter=True)
```

In the diagram, you can see that 'Hens' is related to 'lay' as 'nsubj' which means the nominal subject of the verb lay is 'Hen'. In a similar manner, 'eggs' is related to the verb 'lay' as 'dobj' which means the direct object of the verb, i.e., 'lay' is 'eggs'.

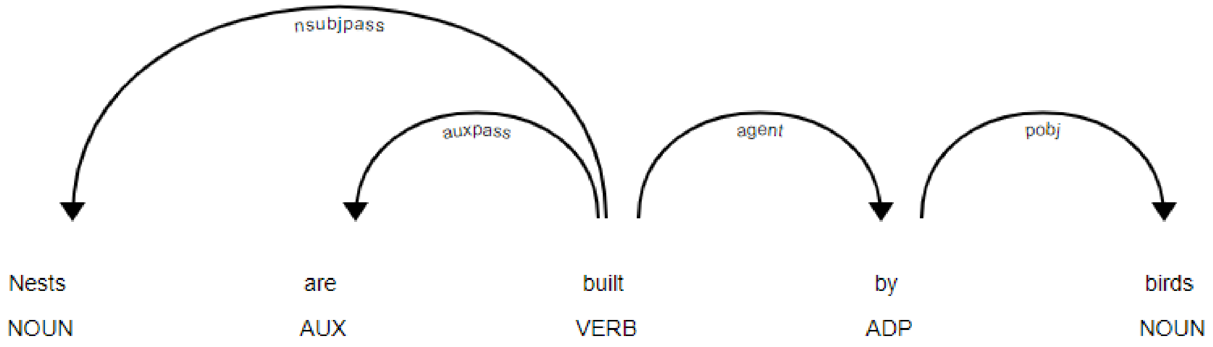
In a similar way, you can create the dependency parse tree of all the sentences which are in 'active' and 'passive' arrays using the following lines of code.

```
#Create the dependency parse tree of all the sentences in the 'active' array.
for sent in active:
    doc = nlp(sent)
    displacy.render(doc, style="dep")

#Create the dependency parse tree of all the sentences in the 'passive' array.
for sent in passive:
    doc = nlp(sent)
    displacy.render(doc, style="dep")
```

Let's look into one of the dependency parse trees of a passive sentence.

Sentence: 'Nests are built by birds'.



You will notice that there is a ‘nsubpass’ relation between a noun and a verb. In the sentence, ‘nests’ is the passive nominal subject of the verb ‘built’. You can read more about ‘nsubpass’ in this link of universal dependencies tags.

When you build the dependency parse tree of all the passive sentences, there is a presence of ‘nsubjpass’ dependency relation most of the time.

Now, can we use this constraint to build a rule to identify a passive sentence? The answer is yes. You are going to build a rule to identify whether a particular sentence is passive or active in the next segment.

Please note that in other versions of SpaCy or Google Colab, you might observe that dependency tags and parse trees are slightly different.

Please note that the number of arrows emitting from a particular word of a sentence in the dependency parse tree are called the ‘children’ of that word. Suppose you have the following sentence:

‘Nests are built by birds’.

Then, the number of children for the word ‘built’ is three, which you can see in the dependency tree of this sentence. You can find the number of children using the following lines of code:

```
import spacy
nlp = spacy.load("en_core_web_sm")
sent = "Nests are built by birds"
doc = nlp(sent)
len(list(doc[3].children))
```

In a similar way, there is the concept of parent. In this sentence, the word ‘built’ is the parent of ‘nests’, ‘are’ and ‘by’ or the word ‘by’ is the parent of the word ‘birds’.

you learnt to create dependency parse trees using displacy to draw those parse trees. You have also seen that the parse tree for an active sentence is quite different from the parse tree of a passive sentence. The difference in the dependency parse tree of an active and a passive sentence can be used to create a rule to identify whether a particular sentence is active or a passive sentence.

So, let's first create a rule to identify the NOUN PoS tag in a given sentence using 'Matcher' class as follows:

```
doc = nlp(passive[0])##add passive[0] sentence to the variable doc
```

```
rule = [{'POS':'NOUN'}]
matcher = Matcher(nlp.vocab)
matcher.add('Rule',[rule])
```

Here, you have defined a dictionary object named 'rule' where the key to the dictionary is 'POS' and the value is NOUN corresponding to the key 'PoS'. In the object 'rule', you have taken the NOUN as the value as you are trying to find out the nouns from the given sentence or 'doc'.

After you have created a dictionary, you will have to create an object named 'matcher' using Matcher class. The Matcher matches the sequences of tokens based on pattern rules. Similar to the example, you have defined a rule to identify the NOUN PoS tag then it will look into the sentence's token and their PoS tags.

Once you have defined the rule and created a 'matcher' object, you need to add this rule, i.e., 'rule' into the 'matcher' object.

Now, you will have to apply the rule on the sentence using the following code to get the positions of the words in a sentence, i.e., 'doc' whose tag is NOUN which you have defined in the rule itself.

```
matcher(doc)
```

Let's try to understand this line of code using the below sentence:

'Eggs are laid by hens'.

There are two nouns 'Eggs' and 'hens' which are located at (0, 1) and (4, 5). This means that the word 'eggs' starts from index 0 and ends before index 1. In a similar way, the word 'hens' starts from index 4 and ends before index 5.

So, once you get the position of the respective NOUN PoS tags, you can apply the following code to get the nouns:

```
doc[0:1]
doc[4:5]
```

Similarly, let’s define the rules for passive voice using the following lines of code:

```
passive_rule = [{'DEP':'nsubjpass'}]
matcher = Matcher(nlp.vocab)
matcher.add('Rule',[passive_rule])
```

Here, you have defined a dictionary ‘passive_rule’ with the key value as ‘DEP’ and the value as ‘nsubjpass’. In the next step, you create an object ‘matcher’ from the ‘Matcher’ class. In the third line, you added the rule ‘passive_rule’ into this ‘matcher’ object to identify the word with ‘nsubjpass’ tag in the given sentence.

Let’s apply this defined rule on ‘doc’ which is passive[0], i.e., ‘Eggs are laid by hens’, to get the location of those tokens whose dependency tag is ‘nsubjpass’ which you have defined in the rule.

```
matcher(doc)
```

The output of the code will be (0, 1).

Let’s try to understand this output using an example. Consider the following sentence:

‘Eggs are laid by hens’.

In this sentence, only the word ‘Eggs’ has an ‘nsubjpass’ tag and it appears at the position of index value 0 and before the index value 1 in the ‘doc’.

Now, let’s define a function to identify whether a function is passive or active.

```
def is_passive(doc,matcher):
    if len(matcher(doc))>0:
        return True
    else:
        return False
```

So, you have defined a very simple function which will return true if there is any value in the ‘matcher(doc)’.This means that this sentence, i.e., ‘doc’ is a passive sentence but it will return false if there are no values in ‘matcher(doc)’ which in turn means that ‘doc’ is an active sentence.

Let’s first apply the defined function on all the active sentences which are present in the array ‘active’.

```
for sent in active:
    doc = nlp(sent)
    print(is_passive(doc,matcher))
```

As there will be no ‘nsubjpass’ tags in the active sentences, the ‘for’ loop will return all values as false for all four active sentences.

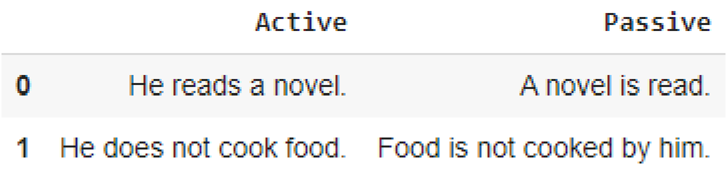
Similarly, when you apply the function on all the passive sentences which are present in the array ‘passive’, you get all values as true for all four passive sentences as there will be a token at the location (0, 1) whose dependency tag will be ‘nsubjpass’.

So, you have learnt how to identify whether a sentence is passive or active by implementing a simple rule.

Parsing – Python Implementation Part - III

You learnt to build a simple rule to identify the passive sentences. In this segment, you will define a more exhaustive rule that would be able to identify most of the passive sentences.

Let’s first look into the data set that Gunnvant has loaded in the dataframe named *‘active_passive’.*



	Active	Passive
0	He reads a novel.	A novel is read.
1	He does not cook food.	Food is not cooked by him.

As you can see in the image, there are two columns in the data set ‘Active’ and ‘Passive’. The ‘Active’ column contains around 40 sentences in active voice and the ‘Passive’ column contains the same sentences corresponding to the column ‘Active’ but in the passive voice.

Let’s understand each step one by one.

- Now, you divided the two columns into two separate data sets *‘active’* and *‘passive’* as follows:

  ```
  active = active_passive['Active']
  passive = active_passive['Passive']
  ```

- Next, you created a rule to identify the passive sentences which you learnt in the previous segment.

  ```
  passive_rule = [{'DEP':'nsubjpass'}]
  matcher = Matcher(nlp.vocab)
  matcher.add('Rule',[passive_rule])
  ```

- Once you created the rule, next you created a function to check whether a sentence is in passive voice or not as follows:

```
def is_passive(doc,matcher):
    if len(matcher(doc))>0:
        return True
    else:
        return False
```

- Let’s apply this rule to check whether a sentence is in active voice or not.

```
cnt = 0
for sent in active:
    doc = nlp(sent)
    if not is_passive(doc,matcher):
        cnt += 1
print(cnt)
```

The output of these lines of code will be ‘40’, which means there are no passive sentences in the ‘active’ dataframe.

- Similarly, apply the rule to check whether a sentence is in passive voice or not.

```
cnt = 0
for sent in passive:
    doc = nlp(sent)
    if is_passive(doc,matcher):
        cnt += 1
print(cnt)
```

The output of these lines of code will be ‘38’, which means there are a total of 38 passive sentences in the ‘passive’ dataframe out of the 40 sentences. This means that there are two sentences in which the defined rule is not valid or applicable.

- Let’s try to investigate those two sentences using the following line of code:

```
cnt = 0
missed = []
for sent in passive:
    doc = nlp(sent)
    if is_passive(doc,matcher):
        cnt += 1
    else:
        missed.append(doc)
print(cnt)
```

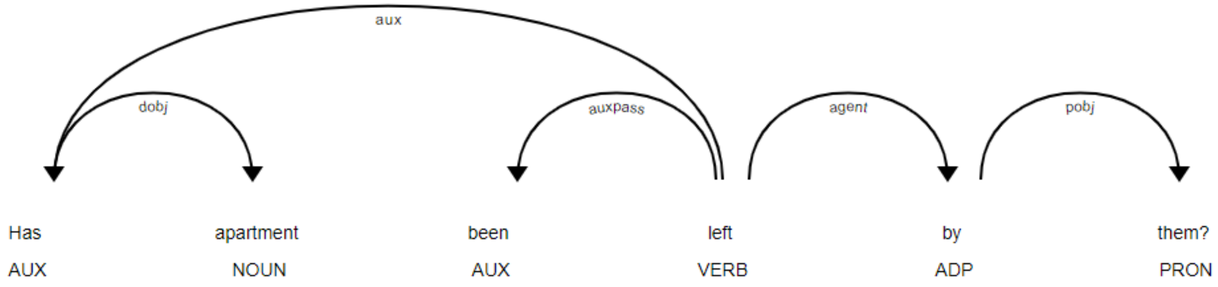
```
missed[0]
missed[1]
```

The two sentences on which the rule was not valid or applicable are as follows:

‘Is a table being bought by Ritika?’
‘Has apartment been left by them?’

- When you visualise the dependency trees of these two sentences, you will find the following diagrams:

```
for doc in missed:
    displacy.render(doc, style="dep")
```



As you can see in the diagram, these two sentences do not have the ‘nsubjpass’ tag. However, they have the ‘auxpass’ tag which can be used to create the rule to identify these sentences as passive sentences. You can refer to the universal dependencies tag in this official link to understand the ‘auxpass’ label. Therefore, we need to update the rule to incorporate these two sentences.

- Let’s define a new rule as follows:

```
passive_rule = [{'DEP':{"IN":['nsubjpass','auxpass']}}]
matcher = Matcher(nlp.vocab)
matcher.add('Rule',[passive_rule])
```

As you can see, a new condition called ‘auxpass’ has been incorporated in the rule.

- Let’s once again check how many sentences have been identified correctly in ‘active’ and ‘passive’ dataframes.

Let’s check the ‘active’ sentences first using the following lines of code:

```
cnt = 0
```

```
for sent in active:
    doc = nlp(sent)
    if not is_passive(doc,matcher):
        cnt += 1
print(cnt)
```

The output of this code will be 40 which correctly identifies all the active sentences as non-passive.

Let’s check the ‘passive’ sentences using the following lines of code:

```
cnt = 0
missed = []
for sent in passive:
    doc = nlp(sent)
    if is_passive(doc,matcher):
        cnt += 1
    else:
        missed.append(doc)
print(cnt)
```

The output of this code will be 40, which correctly identifies all the passive sentences as passive.

So, after improving the rule by adding a dependency tag ‘auxpass’, you get all the passive sentences correctly tagged.

Please note that in other versions of Spacy or Google Colab, you might observe that there is no need to apply the second rule, i.e., ‘nsubjpass’ and ‘auxpass’ and can identify the passive sentences even using the first rule, i.e., ‘nsubjpass’ only.

- Always test your rules and heuristics on a larger corpus to see the effectiveness of the rules.
- One can write intricate matching rules using ‘matcher’ objects.

Summary

NER and CRF

In this session, the following topics have been covered:

1. Name Entity Recognition (NER)
2. Custom NER using Conditional Random Fields (CRF)
3. Python implementation of NER and CRF

Named Entity Recognition

As you already know, with the help of PoS tagging and parsing techniques, you can determine the relationship between the words in a sentence. Now, the next step of understanding the text is Named Entity Recognition (NER).

Consider the following two representations of the given sentence:

'*John bought 300 shares of Apple in 2006'*

Representation1: [John]$_{\textbf{NOUN}}$ bought 300 shares of [Apple]$_{\textbf{NOUN}}$ in [2006]$_{\textbf{NUMBER}}$

Representation2: [John]$_{\textbf{PERSON}}$ bought 300 shares of [Apple]$_{\textbf{ORGANISATION}}$ in [2006]$_{\textbf{DATE}}$

As you can see, Representation1 has tagged the words 'John' and 'Apple' as nouns and '2006' as a number.

On the other hand, Representation 2 indicates the entity of the words like 'John' is tagged as 'PERSON' and 'Apple' as 'ORGANISATION', which provides information about the entities present in the sentence. This output can be achieved by using NER techniques. Essentially, with the help of NER, you will be able to find what the word is referring to like 'John' is a person and 'Apple' is an organisation.

NER techniques are applied in various fields such as search engine, chatbot and mainly in entity extraction in the long texts such as reviews, books, blogs and comments.

Named Entity Recognition (NER) enables you to easily identify the key elements in a piece of text, such as a person's name, locations, brands, monetary values, dates and so on.

Some example sentences with their named-entity recognition are as follows:

Note that GPE is short for geopolitical entity, ORG is short for organisation and PER is short for person.

1. S: 'Why is Australia burning?'
   NER: 'Why is Australia$_{\text{[GPE]}}$ burning?'

2. S: 'UK exits EU'
   NER: 'UK$_{\text{[GPE]}}$ exits EU$_{\text{[ORG]}}$'

3. S: ‘Joe Biden intends to create easier immigration systems to dismantle Trump's legacy’
   NER: ‘Joe Biden$_{[PER]}$ intends to create easier immigration systems to dismantle Trump's$_{[PER]}$ legacy’

4. S: ‘First quarter GDP contracts by 23.9%’
   NER: ‘First quarter$_{[DATE]}$ GDP contracts by 23.9%$_{[PERCENT]}$’

Some commonly used entity types are as follows:

PER: Name of a person (John, James, Sachin Tendulkar)
GPE: Geopolitical entity (Europe, India, China)
ORG: Organisation (WHO, upGrad, Google)
LOC: Location (River, forest, country name)

In this course, we will be using the Spacy toolkit for NER tasks. Spacy contains predefined entity types and pretrained models, which makes our task easier. However, at times these, off-shelf toolkits may not be sufficient for some applications. So, to resolve this issue, you will also learn about some custom NER techniques to build your own NER tagger using the Conditional Random Fields (CRF) model.

Named Entity Recognition How to?

You learnt about the concept of named entities. In this segment, you will learn how the NER system works.

Some important points mentioned as follows:

Noun PoS tags: Most entities are noun PoS tags. However, extracting noun PoS tags is not enough because in some cases, this technique provides ambiguous results.

Let’s consider the following two sentences:

S1: ‘Java is an Island in Indonesia.’

S2: ‘Java is a programming language.’

PoS tagging only identifies ‘Java’ as a noun in both these sentences and fails to indicate that in the first case, ‘Java’ signifies a location and in the second case, it signifies a programming language.

A similar example can be **‘Apple’.** PoS tagging fails to identify that ‘Apple’ could either be an organisation or a fruit.

Let’s take a look at a simple rule-based NER tagger.

Simple rule-based NER tagger: This is another approach to building an NER system. It involves defining simple rules such as identification of faculty entities by searching ‘PhD’ in the prefix of a person's name.

However, such rules are not complete by themselves because they only work on selected use cases. There will always be some ambiguity in such rules.

Therefore, to overcome these two issues, **Machine Learning** techniques can be used in detecting named entities in text.

IOB Labelling

Machine learning can prove to be a helpful strategy in named entity recognition. So, before understanding how exactly ML is used in the NER system, you will learn about IOB labelling and sequence labelling related to the NER system.

IOB (inside-outside-beginning) labeling is one of many popular formats in which the training data for creating a custom NER is stored. IOB labels are manually generated.
This helps to identify entities that are made of a combination of words like ‘Indian Institute of Technology’, ‘New York’ and ‘Mohandas KaramChand Gandhi’.

Suppose you want your system to read words such as ‘Mohandas Karamchand Gandhi', ‘American Express’ and ‘New Delhi’ as single entities. For this, you need to identify each word of the entire name as the PER (person) entity type in the case of, say, ‘Mohandas Karamchand Gandhi'. However, since there are three words in this name, you will need to differentiate them using IOB tags.

The IOB format tags each token in the sentence with one of the following three labels: I - inside (the entity), O - outside (the entity) and B - at the beginning (of entity). IOB labeling can be especially helpful when the entities contain multiple words.

So, in the case of ‘Mohandas Karamchand Gandhi', the system will tag ‘Mohandas’ as B-PER, ‘Karamchand’ as I-PER and ‘Gandhi' as I-PER. Also, the words outside the entity ‘Mohandas Karamchand Gandhi' will be tagged as ‘O’.

Consider the following example for IOB labeling:

Sentence: ‘Donald Trump visit New Delhi on February 25, 2020 ”

Donald	Trump	visit	New	Delhi	on	February	25	,	2020
B-Pers	I-Pers	O	B-GPE	I-GPE	O	B-Date	I-Date	I-Date	I-Date

on	on								

In the example above, the first word of more than one-word entities starts with a **B label,** and the next words of that entity are labelled as **I,** and other words are labelled as **O.**

Note that you will not always find the IOB format only in all applications. You may encounter some other labeling methods as well. So, the type of labelling method to be used depends on the scenario. Let's take a look at an example of a healthcare data set where the labelling contains 'D', 'T', and 'O', which stand for disease, treatment and others, respectively.

S: ‘In[O] the[O] initial[O] stage[O], Cancer[D] can[O] be[O] treated[O] using[O] Chemotherapy[T]’

NER: Python Demonstration

You learnt about named entity types, the benefits of NER tagging over PoS tagging and IOB tags.

In this segment, you will perform a NER demonstration in Python using the Spacy library.

To understand the query, a search engine first tries to find the named entity used in the query, and then corresponding to that entity, it displays the appropriate searches to the user. Named entity recognition is one of the aspects of the search engine mechanism.

You obtained PoS tags of the following sentence:

‘Sumit$_{\text{[PROPN]}}$ is$_{\text{[AUX]}}$ an$_{\text{[DET]}}$ adjunct$_{\text{[ADJ]}}$ faculty$_{\text{[NOUN]}}$ at$_{\text{[ADP]}}$ UpGrad$_{\text{[PROPN]}}$.’

However, PoS tagging failed to distinguish between ‘Sumit’ and ‘UpGrad'.

According to PoS tagging, both ‘Sumit’ and ‘UpGrad’ are proper nouns.

Like PoS tagging, you can also find the named entities present in a sentence using the following code:

```
import spacy # import spacy module
model = spacy.load("en_core_web_sm") #load pre-trained model
doc = "Any sentence"

processed_doc = model(doc); #process input and perform NLP tasks
for ent in processed_doc.ents:
  print(ent.text, " -- ", ent.start_char, " -- ", ent.end_char, " -- ", ent.label_)
```

After processing the model over ‘doc’, we got ‘processed_doc’, which contains an attribute called ‘ents’, which contains all the entities that identify with the Spacy NER system. So, the code above iterates over every token of the sentence and provides the corresponding entity associated with that token that was identified by the Spacy-trained model.

You might have also noticed that the entity types of ‘Sumit’ and Upgrad in the following two sentences are different

‘Sumit is an adjunct faculty of Upgrad$_{[GPE]}$.’

‘Dr. Sumit$_{[PERSON]}$ is an adjunct faculty of Upgrad$_{[ORG]}$’

In the second sentence, the model successfully finds both the entities, but in the first sentence, it fails to identify ‘Sumit’. The reason for this could be that ‘Sumit’ is not present in the corpus on which the model was trained. However, adding ‘Dr’ as a prefix indicates the model that the next word is a person. Based on these observations, we can conclude that there are various situations where systems make errors depending on the application.

Let’s now consider a practical application of NER systems.

Anonymisation of data and redacting personally-identifying information

In many scenarios, we would want to withhold sensitive and confidential information such as names of persons, dates and amounts.

Suppose you are asked to write a program that can anonymise people’s names in many emails for confidential purposes.

For this task, we can use NER techniques to automatically identify PERSONS in the text and remove PERSON names from the text.

Note: we have taken the example of emails from the Enron email data set for illustration in this demo.

- Email source:
http://www.enron-mail.com/email/lay-k/elizabeth/Christmas_in_Aspen_4.html

- Complete Enron data:
http://www.enron-mail.com/

We first identified the person’s name in the email using *‘ent.label== ‘PERSON’.* We then replaced the character of a person’s name with ‘*’ to make it anonymous. Now, based on this anonymisation example, answer the following questions.

You learnt about Name Entity Recognition and how to use the pre-built model in Spacy in Python. In this segment, you will learn about how to look at NER as a sequence labelling problem.

We considered the following two sentences:

‘I drove away in my Jaguar.’
‘The deer ran away seeing the Jaguar.’

The word ‘Jaguar’ in the first sentence refers to a car manufacturing company, and in the second sentence, it refers to a species of animal. Let’s first try to solve the question given below to find the NER tags or entity labels of the word ‘Jaguar’ in these two sentences.

As you can see, both these sentences are assigned the same NER tag, i.e., ORG (organisation) for the word ‘Jaguar’ when you use the predefined NER model or off-the-shelf tools using Spacy.

So, we can conclude that you can not perform Named Entity Recognition using a predefined model in Spacy all the time because you may get the wrong result, as shown in the previous example where you got the correct NER tag for the word ‘Jaguar’ when the word was used as a company’s name, but the word was not not correctly tagged when the word ‘Jaguar’ was used as the name of a species of animal.

Can you recall the PoS tagging using the HMM model approach, which is based on the sequence labelling technique?

The same approach of sequence labelling can be used to perform Name Entity Recognition tasks.

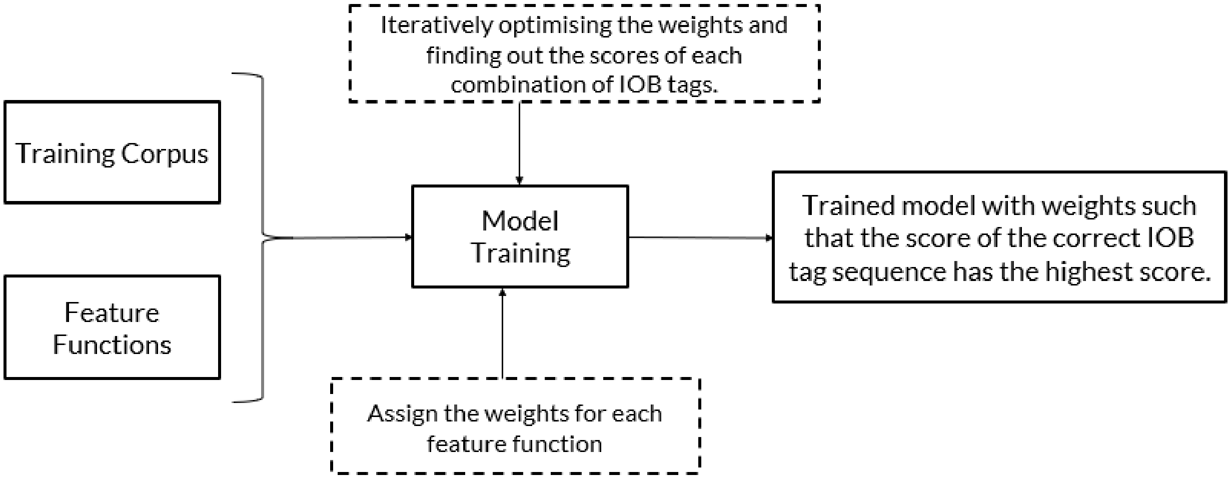
The Conditional Random Field (CRF) can be used as a sequence labelling technique for performing NER tagging. CRF is used to perform custom NER.

Conditional Random Fields are the class of probabilistic models. There are two terms in the CRF nomenclature that you need to keep in mind, which are as follows:

1. Random fields: These indicate that the CRF is probability a distribution-based machine learning model.
2. Conditional: This indicates that the probabilities are conditional probabilities.

To help you understand the logic behind custom NER tagging using the CRF model, we have divided this topic into the following four parts:

1. Overview of custom NER using the CRF model
2. CRF model training
3. Model prediction
4. Python demonstration of custom NER tagging using the CRF model


Iteratively optimising the weights and finding out the scores of each combination of IOB tags.
Training Corpus
Feature Functions
Model Training
Trained model with weights such that the score of the correct IOB tag sequence has the highest score.
Assign the weights for each feature function

Keep this illustration of the overall architecture of the CRF model for custom NER handy. We recommend you to refer to this overview after learning about the CRF model, after which you will be able to better understand the entire process step wi-e.

CRF: Model Training Part - I

In most machine learning model building exercises, you need to build the model using a training data set. Once your model is ready, you test the accuracy of the model or evaluate its performance using the test data set or, in other words, you perform the prediction using the test data set. Similarly, you will train the CRF model using the train data set and get the predictions using the already trained model.

In order to train or build the model, consider the following sentences as the training corpus:

X1= ‘Google Inc. is headquartered in Silicon Valley.’
X2= ‘I went to New York.’

In the training data set, the NER tags such as IOB tags are assigned manually, as shown below.

X1= Google **(B-ORG)** Inc **(I-ORG)** is **(O)** headquartered **(O)** in **(O)** Silicon **(B-LOC)** Valley **(I-LOC)**.

X2= I **(O)** went **(O)** to **(O)** New **(B-LOC)** York **(I-LOC)**.

Where,
ORG and LOC stand for organisation and location, respectively.

For a better understanding of the end-to-end process of training the model, we are considering only two IOB tags, ORG (organisation) and LOC (location).

To simplify, let’s consider only sentence ‘X1’ in the training corpus. For O, ORG and LOC, there can be multiple combinations of IOB tags possible for the model for sentence ‘X1’. Some of them can be written as shown in the table below.

Note that for ‘X1’, we will be using ‘X’ in further discussions.

X	**Y**	**Y’**	**Y”**	**Y”’**
Google	***B-ORG***	B-LOC	B-LOC	B-ORG
Inc	***I-ORG***	I-ORG	I-LOC	I-ORG
is	***O***	O	O	B-ORG
headquartered	***O***	O	O	B-LOC
in	***O***	O	O	B-ORG
Silicon	***B-LOC***	O	B-LOC	B-ORG
Valley	***I-LOC***	O	I-LOC	B-ORG

So, for the input sentence ‘X’, there can be multiple combinations of IOB tags possible, and some of them are shown in the above table. The highlighted one is the correct combination, which is tagged manually in the training data set T**he model needs to train itself or assign the weights in such a way that it assigns the highest score to the correct combination.** You need not worry about what scores or weights are.
Now, apart from the training data set, you require features and their values to train a machine learning model. Here, the training corpus is nothing but the text data. You need to create features out of this text data, which you can feed into the model.

In any machine learning model building task, you need to define the features you want to feed into the model. In custom NER applications, you can define the features using the CRF technique. After defining the features and obtaining their numerical values, you will understand how the model calculates weights and scores.
Some commonly used features in the CRF technique for NER applications are as follows:

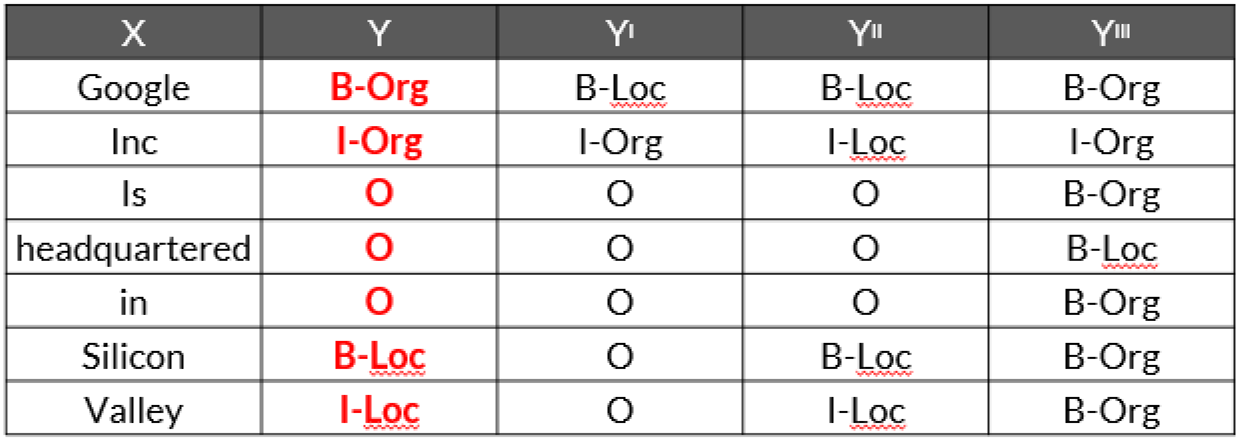
- You can build logic on the input word ‘xi’ and on the surrounding words, which could be ‘xi-1’ or ‘xi+1’.
- The PoS tag of the word ‘xi’ and surrounding words.
- Is a particular word present in a dictionary (dictionary of common names, dictionary of organic chemicals, etc.)
- Word shapes:
    a. 26-03-2021 => dd-dd-dddd
    b. 26 Mar 2021 => dd Xxx dddd
    c. W.H.O => X.X.X
- Presence of prefix and suffixes

You will learn more about feature creation in the subsequent segments.

Also, we have not yet connected the dots regarding scores, weights and features and their values.

CRF: Model Training Part - II

Let’s consider our training data set again.



X	Y	Y^{I}	Y^{II}	Y^{III}
Google	B-Org	B-Loc	B-Loc	B-Org
Inc	I-Org	I-Org	I-Loc	I-Org
Is	O	O	O	B-Org
headquartered	O	O	O	B-Loc
in	O	O	O	B-Org
Silicon	B-Loc	O	B-Loc	B-Org
Valley	I-Loc	O	I-Loc	B-Org

The first part of any model building activity is to have a training data set. We have a training data set in which each token is manually tagged. The correct tags are shown in red in the table above. There can be many possible combinations for the ORG, LOC and O tags for the given training data set, which are also shown in the table above.

The next part of the model building activity is to have some features to feed into the model. To get some features for this data set, it is necessary to define the feature functions for this data set.

So, we defined three feature functions for this example, which are as follows:

f1 (X, xi, xi-1, i) = 1 if xi= Xx+; otherwise, 0 (Words starting with an uppercase letter)
f2 (X, xi, xi-1, i) = 1 if xi= Noun and xi-1 is Noun; otherwise, 0 (Continuous entity)
f3 (X, xi, xi-1, i) = 1 if xi = Inc and xi-1 = B-Org; otherwise, 0 (Company names often end with Inc.)

We can elaborate the definition of the feature functions as follows:

- The f1 feature indicates that if a particular word in the given sentence starts with an uppercase letter, then assign 1 as the value of f1; otherwise, assign 0 as the value of f1 to this word.
- The f2 feature indicates that if a particular word in the given sentence has a PoS tag of noun and the word before it also has a PoS tag of noun, then assign 1 as the value of f2 as to this word; otherwise, assign 0 as the value of f2 to this word.

- The f3 feature indicates that if a particular word in the given sentence is 'Inc' and the word before it has the NER tag of B-ORG, then assign 1 the value of f3 to this word; otherwise, assign 0 as the value of f3 to this word.

So, the model considers all the possible combinations of IOB tags of the given training example and calculates the scores of each combination using the weights corresponding to each feature function. The model starts the computation by taking any random initial weights for each feature function and iteratively modifies the weights until it reaches a stage where the score of the correct IOB tag sequence is the highest.

Note that we are not covering the details of how the model optimises the weights and calculates the final values of the weights. However, the reasoning is quite similar to the gradient descent algorithm. You can refer to this link to read about the workings of the CRF algorithm.

LINK
https://homepages.inf.ed.ac.uk/csutton/publications/crftut-fnt.pdf
https://medium.com/data-science-in-your-pocket/named-entity-recognition-ner-using-conditional-random-fields-in-nlp-3660df22e95c#:~:text=CRF%20is%20amongst%20the%20most,denoted%20by%20y%E1%B5%A2%E2%82%8B%E2%82%81).

Let's consider the weights as w1, w2 and w3, corresponding to the features f1, f2 and f3, respectively.

Let's first calculate the values of all three features for the Y' combination of IOB tags.

X= Google **(B-LOC)** Inc **(I-ORG)** is **(O)** headquartered **(O)** in **(O)** Silicon **(O)** Valley **(O)**.

- f1 (X, xi, xi-1, i) = 1 if xi= Xx+; otherwise, 0 (words starting with an uppercase letter)

		f1
Google	B-LOC	**1 (because 'Google' starts with an uppercase**

		letter)
Inc	I-ORG	**1 (because ‘Inc’ starts with an uppercase letter)**
is	O	**0**
headquartered	O	**0**
in	O	**0**
Silicon	O	**1 (because ‘Silicon’ starts with an uppercase letter)**
Valley	O	**1 (because ‘Valley’ starts with an uppercase letter)**

- f2 (X, xi, xi-1, i) = 1 if xi= Noun and xi-1 is Noun; otherwise, 0 (Continuous entity)

		f2
Google	B-LOC	**0**
Inc	I-ORG	**0**
is	O	**0**
headquartered	O	**0**
in	O	**0**
Silicon	O	**0**
Valley	O	**1 (because ‘Valley’ is a noun and ‘Silicon is also noun)**

- f3 (X, xi, xi-1, i) = 1 if xi = Inc and xi-1 = B-Org; otherwise, 0 (Company names often end with Inc)

		f3
Google	B-LOC	**0**

Inc	I-ORG	**0 (because the previous word 'Google' does not have the B-ORG tag)**
is	O	**0**
headquartered	O	**0**
in	O	**0**
Silicon	O	**0**
Valley	O	**0**

CRF: Model Training Part - III

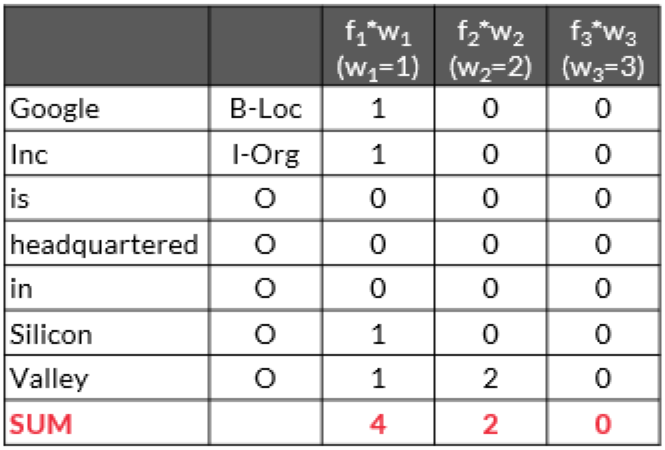
Let's consider the final optimised weights calculated by the model after many iterations are w1, w2 and w3, corresponding to the features f1, f2 and f3, respectively, and the values of optimised w1, w2 and w3 are 1, 2 and 3, respectively.

We will calculate the values of scores corresponding to each of the combinations of NER tags after the model has iteratively optimised the weights as w1 = 1, w2 = 2 and w3 = 3, corresponding to the features f1, f2 and f3, respectively.

you learnt how to calculate scores.

Let's consider the Y' combination of the NER tags for the given training data set.

X= Google **(B-LOC)** Inc **(I-ORG)** is **(O)** headquartered **(O)** in **(O)** Silicon **(O)** Valley **(O)**.



		f_1*w_1 (w_1=1)	f_2*w_2 (w_2=2)	f_3*w_3 (w_3=3)
Google	B-Loc	1	0	0
Inc	I-Org	1	0	0
is	O	0	0	0
headquartered	O	0	0	0
in	O	0	0	0
Silicon	O	1	0	0
Valley	O	1	2	0
SUM		4	2	0

As we have already discussed, the model optimises the weights corresponding to each of the features f1, f2 and f3, and the values of optimised w1, w2 and w3 are 1, 2 and 3, respectively.

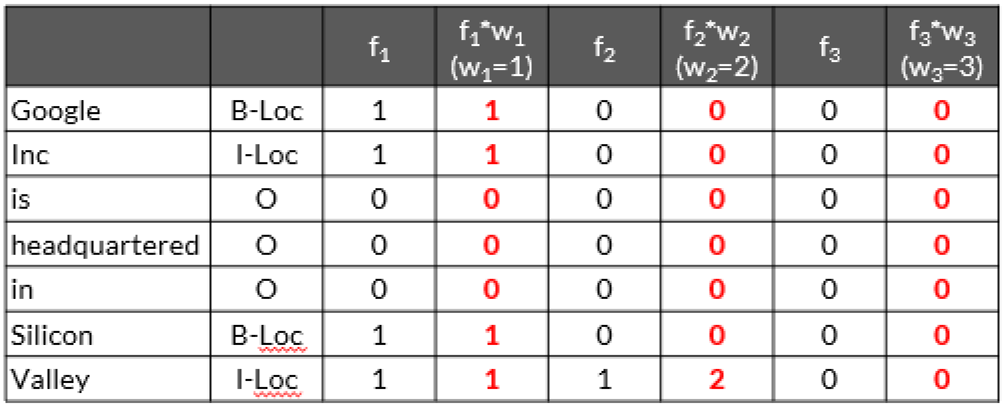
So, the value of the score for this combination of NER tags will be 4+2+0 = 6, which you will obtain after multiplying each feature function values with their corresponding weights, as shown in the table above.

Now, let’s consider another combination of Y’’ and Y of the NER tags for the training sentence ‘X’

First, you need to calculate the score for the combination Y’’ of the NER tags for the training sentence ‘X’, as shown below.

X= Google **(B-LOC)** Inc **(I-LOC)** is **(O)** headquartered **(O)** in **(O)** Silicon **(B-LOC)** Valley **(I-LOC)**.

The calculated score using the optimised weights for this combination will be as shown in the table below.

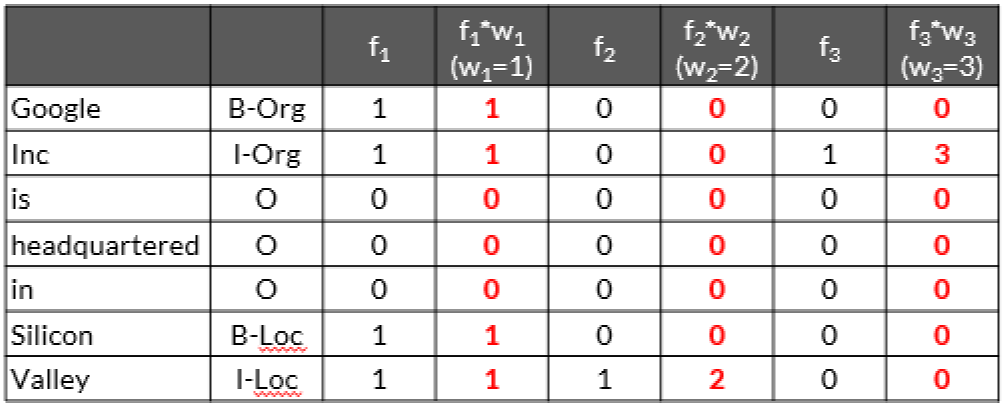


		f_1	f_1*w_1 (w_1=1)	f_2	f_2*w_2 (w_2=2)	f_3	f_3*w_3 (w_3=3)
Google	B-Loc	1	1	0	0	0	0
Inc	I-Loc	1	1	0	0	0	0
is	O	0	0	0	0	0	0
headquartered	O	0	0	0	0	0	0
in	O	0	0	0	0	0	0
Silicon	B-Loc	1	1	0	0	0	0
Valley	I-Loc	1	1	1	2	0	0

Score = 4 + 2 + 0 = 6
Similarly, you need to calculate the score for the combination Y (the correct sequence of NER tags) of the NER tags for the training sentence ‘X’, as shown below.

X= Google **(B-LOC)** Inc **(I-LOC)** is **(O)** headquartered **(O)** in **(O)** Silicon **(B-LOC)** Valley **(I-LOC)**.

The calculated score using the optimised weights for this correct combination will be as shown in the table below.



		f_1	f_1*w_1 (w_1=1)	f_2	f_2*w_2 (w_2=2)	f_3	f_3*w_3 (w_3=3)
Google	B-Org	1	1	0	0	0	0
Inc	I-Org	1	1	0	0	1	3
is	O	0	0	0	0	0	0
headquartered	O	0	0	0	0	0	0
in	O	0	0	0	0	0	0
Silicon	B-Loc	1	1	0	0	0	0
Valley	I-Loc	1	1	1	2	0	0

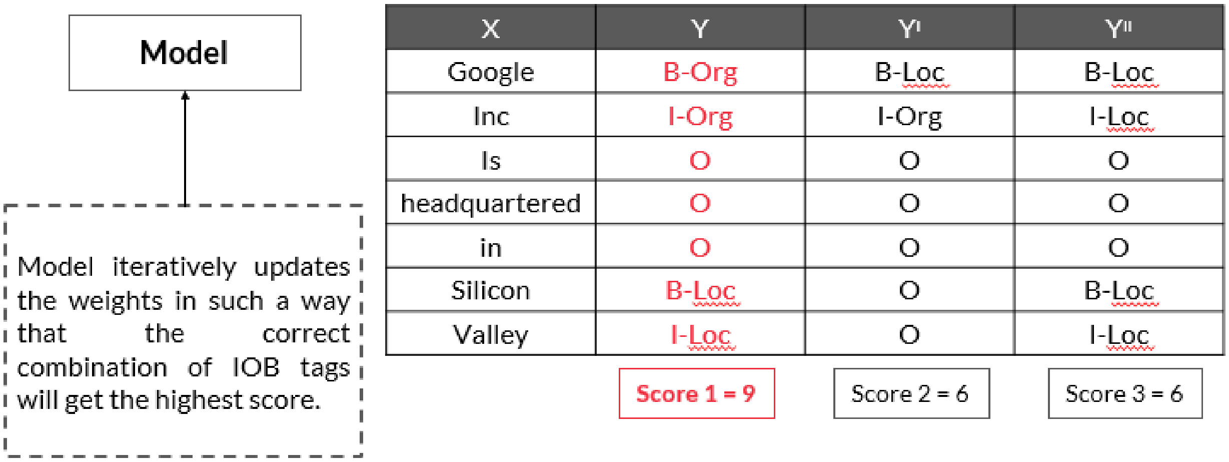
Score = 4 + 2 + 3 = 9

As you can observe, the score of the correct NER tag sequence will be 9, which is the maximum among all the combinations that we have considered.

To understand the CRF intuitively, we are considering the values of w1, w2 and w3 as 1, 2 and 3, respectively. We arrived at these values after optimising the weights. Using these weights, our model obtains the maximum score of 9 for the correct NER tag sequence.

It is important to note that the model finally calculates the scores for all the combinations, and the weights are chosen in such a way that the model calculates the highest score for the correct labelling sequence.

As a summary point you can keep the following image:



Model iteratively updates the weights in such a way that the correct combination of IOB tags will get the highest score.

X	Y	Y'	Y''
Google	B-Org	B-Loc	B-Loc
Inc	I-Org	I-Org	I-Loc
Is	O	O	O
headquartered	O	O	O
in	O	O	O
Silicon	B-Loc	O	B-Loc
Valley	I-Loc	O	I-Loc
	Score 1 = 9	Score 2 = 6	Score 3 = 6

Score1 = 9 is the highest.

So far, we have defined features and trained the model using the training corpus by optimising the weights corresponding to each defined feature function.

CRF: Model Prediction

You learnt how the model makes predictions for a given test data set.

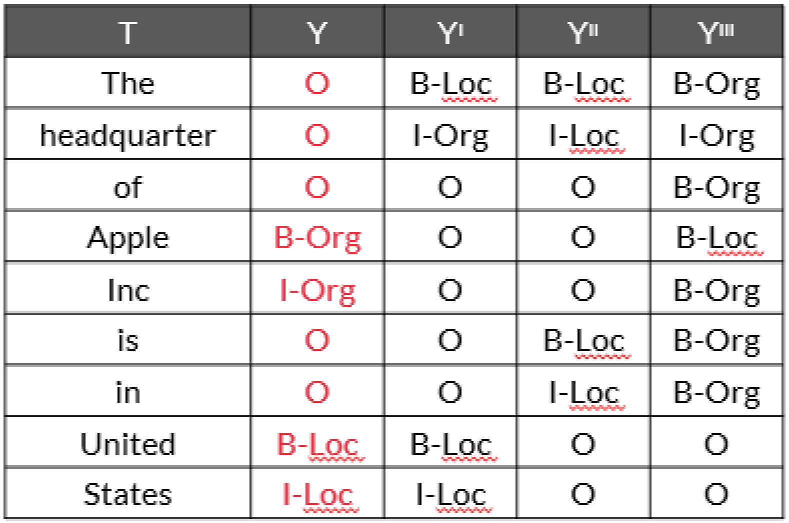
To understand the prediction part, let’s consider the following test sentence ‘T’:

T= ‘The headquarter of Apple Inc is in United States.’

The correct NER tags for this sentence will be as follows:

T= ‘The **(O)** headquarter **(O)** of **(O)** Apple **(B-ORG)** Inc **(I-ORG)** is **(O)** in **(O)** United **(B-LOC)** States **(I-LOC)**.’

Ideally, the pre-trained CRF model should predict these correct NER tags. As you already know, the model will have many possible combinations for this test sentence. Some of the possible combinations are given in the table given below.



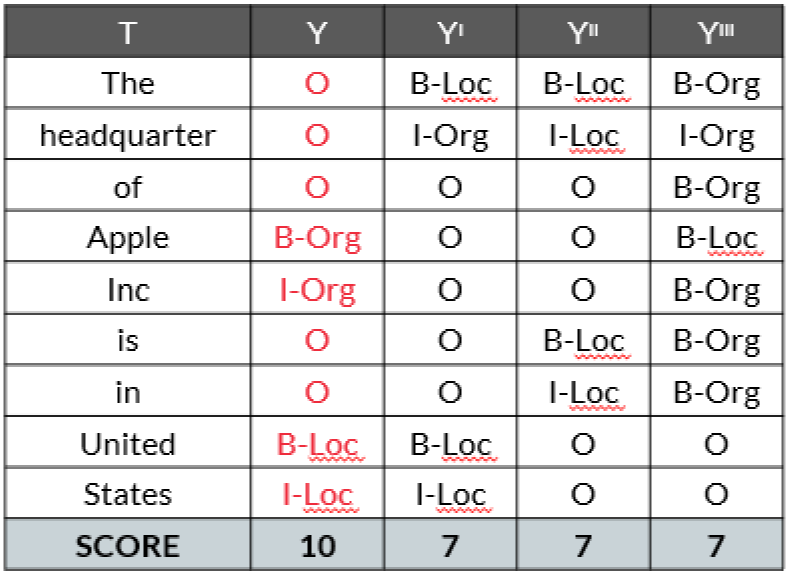
T	Y	Y^{I}	Y^{II}	Y^{III}
The	O	B-Loc	B-Loc	B-Org
headquarter	O	I-Org	I-Loc	I-Org
of	O	O	O	B-Org
Apple	B-Org	O	O	B-Loc
Inc	I-Org	O	O	B-Org
is	O	O	B-Loc	B-Org
in	O	O	I-Loc	B-Org
United	B-Loc	B-Loc	O	O
States	I-Loc	I-Loc	O	O

Where,
The red values represent the correct combinations for ‘T’.

So, the model calculates the scores for each possible combination of NER tags for a given input ‘T’ using the already trained weights and provides the output with an NER sequence that has the highest score.

If our model is trained correctly, then it should generate the highest score for the correct NER sequence using the trained weights.

Let’s take a look at the table given below.



T	Y	Y^{I}	Y^{II}	Y^{III}
The	O	B-Loc	B-Loc	B-Org
headquarter	O	I-Org	I-Loc	I-Org
of	O	O	O	B-Org
Apple	B-Org	O	O	B-Loc
Inc	I-Org	O	O	B-Org
is	O	O	B-Loc	B-Org
in	O	O	I-Loc	B-Org
United	B-Loc	B-Loc	O	O
States	I-Loc	I-Loc	O	O
SCORE	**10**	**7**	**7**	**7**

As you can observe, out of the many combinations possible, we have taken only three combinations. The calculated scores using the trained weights (w1=1, w2=2 and w3=3) for these possible combinations will be 10, 7, 7 and 7, as shown in the table above. The highest score (10) is calculated for the correct NER sequence (highlighted in the red).

This is how the model predicts the sequence of NER tags for a given input. If the model calculates the highest score for a different incorrect combination of NER sequence, then it will predict that incorrect sequence as the output, and the model will throw an error.

Now, you have understood the end-to-end process of CRF model training and prediction.

Custom NER: Python Implementation Part-I

You have learnt how the CRF model works for custom NER applications. Now, let's begin the hands-on part of the CRF model using Python.

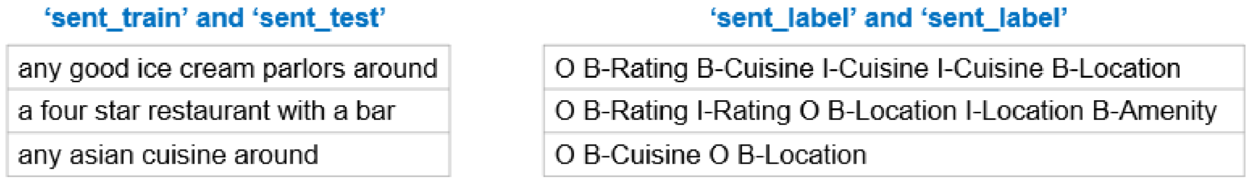
So, the data set is around restaurant reviews. In total, the following four data sets have been provided to you:

1. ***'sent_train'***
2. ***'sent_test'***
3. ***'label_train'***
4. ***'label_test'***

The *'sent_train'* and the 'sent_test' contain the sentences, and a new line is a new sentence in both these two files.

The *'label_train'* and *'label_test'* contain the corresponding NER labels or tags for each token present in each sentence of the *'sent_train'* and *'sent_test'* files.

So, there is a one-to-one mapping of the sentences that are present in the 'sent_train' and the 'sent_test' files, with the NER labels or tags that are present in the 'label_train' and 'label_test' files. You can understand the structure of these files with the help of the image given below.



'sent_train' and 'sent_test'	'sent_label' and 'sent_label'
any good ice cream parlors around	O B-Rating B-Cuisine I-Cuisine I-Cuisine B-Location
a four star restaurant with a bar	O B-Rating I-Rating O B-Location I-Location B-Amenity
any asian cuisine around	O B-Cuisine O B-Location

This data set contains approximately eight NER tags or labels that are available in the train and the test data sets; you can list these in the following way.

Rating, Amenity, Location, Restaurant_Name, Price, Hours, Dish, Cuisine.

So, once you have understood the data set structure, the next task in the CRF model building activity is to define the features.
Sumit installed two very important packages, which are given below:

1. pycrf
2. sklearn-crfsuite

Through these packages, you perform the CRF model building tasks.

Sumit created an end-to-end function to define some of the features. Now, let's understand the features one by one in the below function.

```
#Define a function to get the above defined features for a word.
def getFeaturesForOneWord(sentence, pos):
  word = sentence[pos]

  features = [
    'word.lower=' + word.lower(), # serves as word id
    'word[-3:]=' + word[-3:],     # last three characters
    'word[-2:]=' + word[-2:],     # last two characters
    'word.isupper=%s' % word.isupper(),  # is the word in all uppercase
    'word.isdigit=%s' % word.isdigit(),  # is the word a number
    'words.startsWithCapital=%s' % word[0].isupper() # is the word starting with a capital letter
  ]

  if(pos > 0):
    prev_word = sentence[pos-1]
    features.extend([
    'prev_word.lower=' + prev_word.lower(),
    'prev_word.isupper=%s' % prev_word.isupper(),
    'prev_word.isdigit=%s' % prev_word.isdigit(),
  'prev_words.startsWithCapital=%s' % prev_word[0].isupper()
 ])
 else:
  features.append('BEG') # feature to track begin of sentence

 if(pos == len(sentence)-1):
  features.append('END') # feature to track end of sentence

 return features
```

This function is divided into the following two parts:

1. Getting the information of the current word
2. Getting the information of the previous word

So, the function ***'getFeaturesForOneWord(sentence, pos)'*** takes two inputs: one is 'sentence' and the other is 'pos', which is the position in the input sentence (the word appears at that position) for which you are going to get the feature values.

Suppose, you apply this function ***'getFeaturesForOneWord(sentence, pos)***' using the following inputs:

sentence: "a place that serves soft serve ice cream"
pos: 3

So, using this input, the function ***'getFeaturesForOneWord(sentence, pos)***' will be applied on the word ***'that'*** that is appearing at position 3 in the given sentence.

So, in the first of the function '***getFeaturesForOneWord(sentence, pos)***', you are defining the features to get the information on the current word.

Let's understand the next code as below.

```
features = [
  'word.lower=' + word.lower(), # serves as word id
  'word[-3:]=' + word[-3:],      # last three characters
  'word[-2:]=' + word[-2:],      # last two characters
  'word.isupper=%s' % word.isupper(),  # is the word in all uppercase
  'word.isdigit=%s' % word.isdigit(),  # is the word a number
  'words.startsWithCapital=%s' % word[0].isupper() # is the word starting with a capital letter
]
```

Here, you can see the six feature functions, which are as follows:

- Feature-1 ('word.lower'): This functionality is converting the current word into lowercase and giving the value of feature as a lowercase word.
- Feature-2 ('word[-3:]'): This feature is getting the last three characters of the current word. The purpose of defining this feature is to check whether a word contains 'ing' or 'ed' or any other characters at the end so that the model can check if it is a verb or not.
- Feature-3 ('word[-2:]'): This feature is getting the last two characters of the current word. The purpose of defining this feature is to check whether a word contains 'ing', 'ed' or any other characters at the end so that the model can check if it is a verb or not.

- Feature-4 (‘word.isupper’): This feature will return True if the word is in uppercase; otherwise, it will return False.
- Feature-5 (‘word.isdigit’): This feature will return True if the word is a digit; otherwise, it will return False.
- Feature-6 (‘words.StartsWithCapital’): This feature will return True if the word starts with an uppercase character; otherwise, it will return False.

It is very important to look into the word preceding the current word so that the CRF model can learn more features to predict the result accurately. So, in the next part of the function ***‘getFeaturesForOneWord(sentence, pos)’***, you will define four features to get the information of the preceding word of the current word.

- Feature-7 (‘prev_word.lower’): This function converts the previous word into lowercase and gives the value of feature as a lowercase word.
- Feature-8 (‘prev_word.isupper’): This feature will return True if the previous word is in uppercase; otherwise, it will return False.
- Feature-9 (‘prev_word.isdigit’): This feature will return True if the previous word is a digit; otherwise, it will return False.
- Feature-10 (‘prev_words.StartsWithCapital’): This feature will return True if the previous word starts with an uppercase character; otherwise, it will return False.

Now, knowing the position of the word is preferable, whether it appears at the beginning or at the end of the sentence. For this, you have defined two more features as shown below:

- Feature-11 (‘BEG’): This feature will return ‘BEG’ if the word is the first word of the sentence.
- Feature-12 (‘END’): This feature will return ‘END’ if the word is the last word of the sentence.

So, you have defined some of the features as discussed above in this use case.

Once you have defined the feature functions, in this section, you learnt how to compute the features values.

```
# Define a function to get features for a sentence
# using the 'getFeaturesForOneWord' function.

def getFeaturesForOneSentence(sentence):
  sentence_list = sentence.split()
  return [getFeaturesForOneWord(sentence_list, pos) for pos in range(len(sentence_list))]
```

Here, in this function, you are getting the features for the sentence using the already defined ***‘getFeaturesForOneWord’*** function. So, first, you split the sentence into words and store it into a list object named ***‘sentence_list’***. Once you get the splitted words, you apply the ***‘getFeaturesForOneWord’*** function to each word of the sentence.

Similarly, split the labels into a list using the following lines of code:

```
# Define a function to get the labels for a sentence.
def getLabelsInListForOneSentence(labels):
```

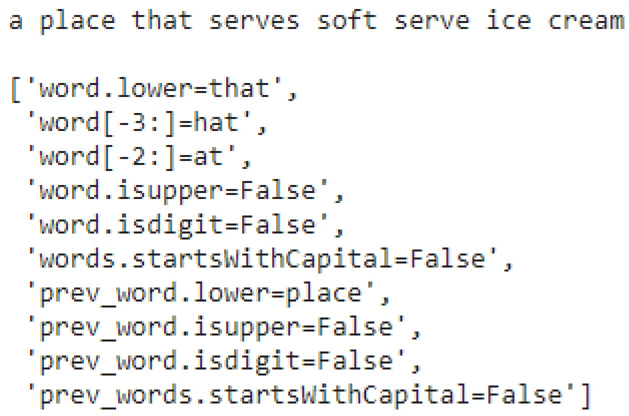
return labels.split()

Now, let’s compute the features for the sixth sentence in the *‘sent_train’* data set. So, the sixth sentence of this data set is given below:

“a place that serves soft serve ice cream”

First, apply the ***‘getFeaturesForOneSentence’*** function on this sentence and then focus on the third word of the sentence, i.e., ***‘that’***.

When you print the computed feature values for the word ***‘that’***, you get the following output.



```
a place that serves soft serve ice cream

['word.lower=that',
 'word[-3:]=hat',
 'word[-2:]=at',
 'word.isupper=False',
 'word.isdigit=False',
 'words.startsWithCapital=False',
 'prev_word.lower=place',
 'prev_word.isupper=False',
 'prev_word.isdigit=False',
 'prev_words.startsWithCapital=False']
```

Since this word does not occur at the start or at the end of the sentence, it contains no value such as ‘BEG’ or ‘END’.

Next, you need to define the input and the output for the train and the test data to feed into the CRF model.

X_train = [getFeaturesForOneSentence(sentence) for sentence in train_sentences]
Y_train = [getLabelsInListForOneSentence(labels) for labels in train_labels]

X_test = [getFeaturesForOneSentence(sentence) for sentence in test_sentences]
Y_test = [getLabelsInListForOneSentence(labels) for labels in test_labels]

So, the *‘X_train’* and the *‘X_test’* are working as the input variables, and the *‘Y_train’* and the *‘Y_test’* are working as the output variable for our model.

The input variable (X) is computed using the function ***‘getFeaturesForOneSentence’***, and the output variable (Y) is computed using the ***‘getLabelsInListForOneSentence’*** function.

Now, you have feature values for each word in all the sentences in the train and the test data sets as the input ‘X’ and the labels the the output ‘Y’.

Custom NER: Python Implementation Part-II

You learnt how to build the CRF model and then evaluate it.

Model building and evaluation are very easy to understand.
Before building the model, you need to import the following libraries:

```
import sklearn_crfsuite
from sklearn_crfsuite import metrics
```

Next, you can build the model using the following lines of code:

```
# Build the CRF model.
crf = sklearn_crfsuite.CRF(max_iterations=100)
crf.fit(X_train, Y_train)
```

Here, you fit the model using *'X_train'* and *'Y_train'*. Sumit took 100 iterations, but you can modify it as per your requirement.

Sumit evaluated and calculated the f1 score using the following lines of code:

```
# Calculate the f1 score using the test data
Y_pred = crf.predict(X_test)
metrics.flat_f1_score(Y_test, Y_pred, average='weighted')
```

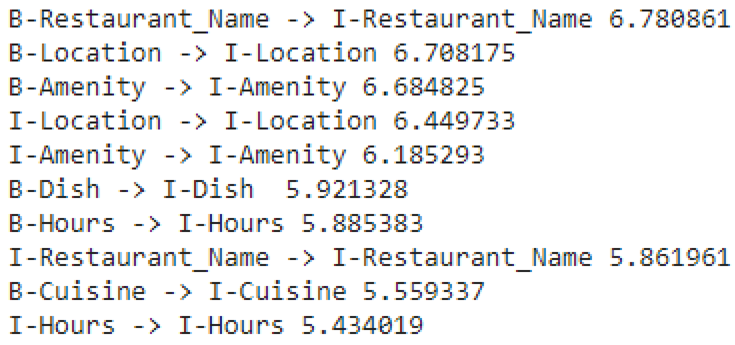
The calculated f1 score is 0.8744, which is good, and we can deduce that the model is performing very well.

Now, you will learn another aspect of CRF, which is how the model is learning from the training data set.

Run the following code.

```
print_top_likely_transitions(crf.transition_features_)
```

The output will be as follows.



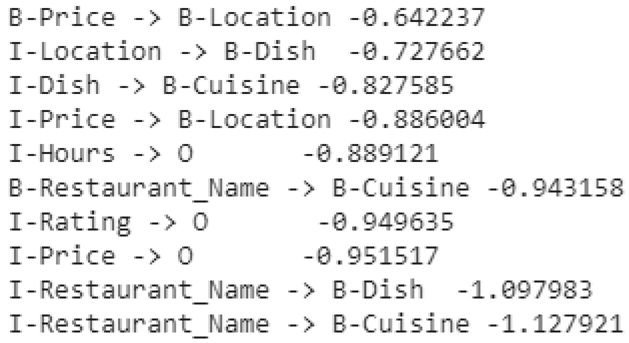
```
B-Restaurant_Name -> I-Restaurant_Name 6.780861
B-Location -> I-Location 6.708175
B-Amenity -> I-Amenity 6.684825
I-Location -> I-Location 6.449733
I-Amenity -> I-Amenity 6.185293
B-Dish -> I-Dish  5.921328
B-Hours -> I-Hours 5.885383
I-Restaurant_Name -> I-Restaurant_Name 5.861961
B-Cuisine -> I-Cuisine 5.559337
I-Hours -> I-Hours 5.434019
```

The output indicates that the score of occurrence of ‘I-Restaurent’ immediately after the ‘B-Restaurent’ is 6.78, and so on. It is intuitive and clear that the probability of the occurrence of the ‘I-Restaurent’ label immediately after the ‘B-Restaurent’ label is higher.

Run the following code.

print_top_unlikely_transitions(crf.transition_features_)

The output will be as follows.



```
B-Price -> B-Location -0.642237
I-Location -> B-Dish  -0.727662
I-Dish -> B-Cuisine -0.827585
I-Price -> B-Location -0.886004
I-Hours -> O       -0.889121
B-Restaurant_Name -> B-Cuisine -0.943158
I-Rating -> O       -0.949635
I-Price -> O       -0.951517
I-Restaurant_Name -> B-Dish  -1.097983
I-Restaurant_Name -> B-Cuisine -1.127921
```

The output indicates that the score of occurrence of ‘B-Location’ immediately after ‘B-Price’ is -0.64, and so on. It is intuitive and clear that the probability of the occurrence of the ‘B-Location’ label immediately after the ‘B-Price’ label is quite low.

So, you have completed the practical implementation of the custom NER using the CRF model.