Lecture Notes

Syntactic Processing

Syntactic processing is widely used in applications such as question answering systems, information extraction, sentiment analysis, grammar checking etc. In this module, you learnt that there are three broad levels of syntactic processing - (Parts-of-Speech) POS tagging, constituency parsing, and dependency parsing. And, POS tagging is a crucial task in syntactic processing and is used as a **preprocessing step** in many NLP applications.

Parts-of-Speech Tags

You learnt about most commonly used tags in Penn Treebank of NLTK. Here is the list of the commonly used tags:

Class	Tag	Part-of-Speech	Definition	Examples
Determiner	DT	Determiner	Describes a reference to a noun	This, that, the, a
Noun	NN	Noun, singular or mass	tag of singular forms of common nouns	Cat, box, animal
	NNS	Noun, singular or mass	tag of plural forms of common nouns	Cats buses, animals
	NNP	Proper noun, singular	Tag of noun names, places and things	India, Rahul, Eric
	NNPS	Proper noun, plural	Names of nations and nationalities, Personal names	Germans, Indians, three *Billys*
Conjunction	CC	Coordinating Conjunctions	Connects words, phrases, and clauses	And but, or

Verb	VB	Verb, base form	Verbs in the base form	Learn, eat, study
	VBD	Verb, past form	Past tense verbs to express action/state of the past	Learnt, ate, studied
	VBG	Verb, gerund or present participle	Verbs ending with '-ing'	Dying, lying, travelling
	VBN	Verb, past participle	Verbs in the past form when used with has/have/had/modal verb. Common scenarios are: Have + 'verb' Might + have + 'verb' were/is + 'verb'	- Have *died* - Had *waited* -Could have *written* - were *excited*
	VBP	Verb, non-3rd person singular present	Non 3rd person verb to express routines/habits, facts/truth and thoughts and feelings (used without word 'to')	I *like* this game. I *go* for a walk daily
	VBZ	Verb, 3rd person singular present	Verbs that are used for 3rd person singular entities -ends with 's', 'es'	Argues, catches, replies
Adjective	JJ	Adjective	Words describing noun	Tall, large, generous
Adverb	RB	Adverb	Words modifying a verb, adjective or other adverb	Now, first, *Slowly* went

Preposition	IN	Preposition	Links nouns, pronouns and phrases to other words in a sentence	Above, across, in, near of
Pronoun	PRP	Personal Pronoun	Substitutes of noun	I, he, she, they

Different Approaches to POS Tagging

Next, you learnt about the four main techniques used for POS tagging:

- **Lexicon-based** approach uses the following simple statistical algorithm: for each word, it assigns the POS tag that most frequently occurs for that word in some training corpus.
  For example, it will assign the tag "verb" to any occurrence of the word "run" if "run" is used as a verb more often than any other tag.
- **Rule-based** taggers first assign the tag using the lexicon method and then apply predefined rules. Some examples of rules are: Change the tag to VBG for words ending with '-ing', Changes the tag to VBD for words ending with '-ed', etc.

You also learnt to implement the lexicon and rule-based tagger on the Treebank corpus of NLTK. Next, you learnt about:

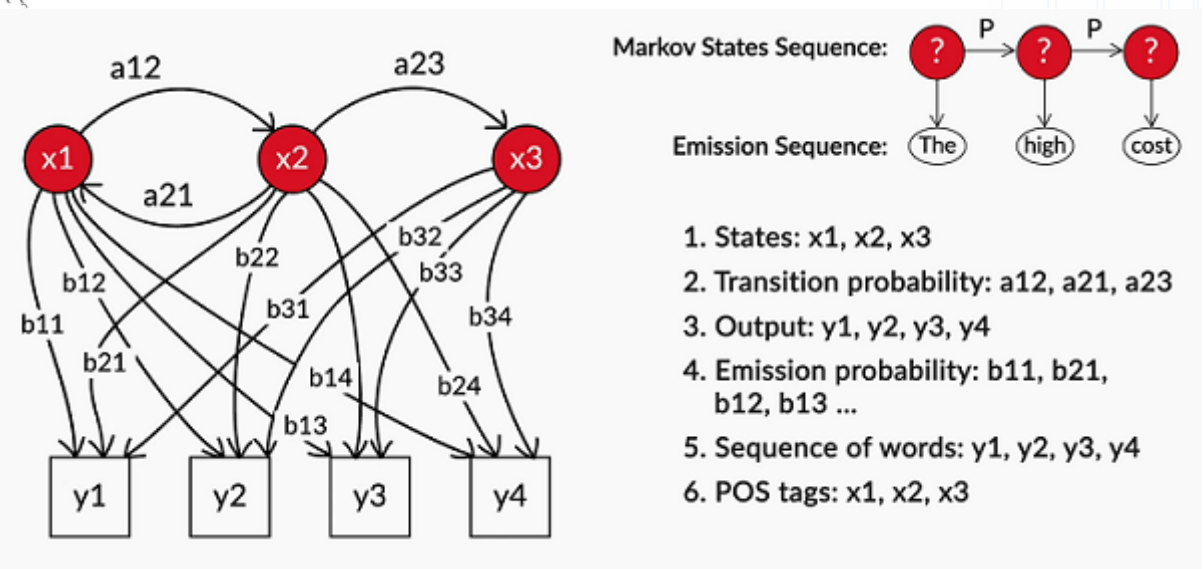
- **Probabilistic (or stochastic)** techniques don't naively assign the highest frequency tag to each word, instead, they look at slightly longer parts of the sequence and often use the tag(s) and the word(s) appearing before the target word to be tagged. You learnt about the commonly used probabilistic algorithm for POS tagging - the *Hidden Markov Model (HMM)*

Deep-learning based POS tagging: Recurrent Neural Networks (RNNs) are used for sequential modeling processes. In this module, you got a basic overview on how RNNs are used for POS tagging.

Hidden Markov Models

In probabilistic method, you learnt about Markov processes and HMMs. **Markov processes** are commonly used to model sequential data, such as text and speech. You learnt that the first-order Markov assumption states that the probability of an event (or state) depends only on the previous state.

The **Hidden Markov Model,** an extension to the Markov process, is used to model phenomena where the states are hidden and they emit observations. The transition and the emission probabilities specify the probabilities of transition between states and emission of observations from states, respectively. In POS tagging, the states are the POS tags while the words are the observations. To summarise, a Hidden Markov Model is defined by the initial state, emission, and the transition probabilities. Refer to following image to revise transition, emission probabilities:



You learnt how to calculate the probability of a tag sequence for a given sequence of words. The POS tag T_i for given word W_i depends on two things: POS tag of the previous word and the word itself.
$P(T_i | W_i) = P(W_i|T_i) * P(T_{i-1}|T_i)$

So, the probability of a tag sequence (T_1, T_2, T_3, …, T_n) for a given the word sequence (W_1, W_2, W_3, …, W_n) can be defined as:
$P(T|W) = (P(W_1|T_1) * P(T_1|start)) * (P(W_2|T_2) * P(T_2|T_1)) * ... * (P(W_n|T_n) * P(T_n|T_{n-1}))$

You learnt that for a sequence of n words and t tags, a total of $\mathbf{t}^n$ tag sequences are possible. The Penn Treebank dataset in NLTK itself has 36 POS tags, so for a sentence of length say 10, there are 36^{10} possible tag sequences.

Viterbi Heuristic

Next, you studied how **Viterbi Heuristic** can deal with this problem by taking a *greedy* approach. The basic idea of the **Viterbi algorithm** is as follows - given a list of observations (words) $O_1, O_2 .... O_n$ to be tagged, rather than computing the probabilities of all possible tag sequences, you assign tags sequentially, i.e. assign the most likely tag to each word using the previous tag.

More formally, you assign the tag T_i to each word W_i such that it **maximises the likelihood**:
$P(T_i | W_i) = P(W_i|T_i) * P(T_{i-1}|T_i)$,

where T_{i-1} is the tag assigned to the previous word. The probability of a tag T_i is assumed to be **dependent only on the previous tag** T_{i-1}, and hence the term $P(T_i|T_{i-1})$ - *Markov Assumption.*

Next you learnt that the Viterbi algorithm is an example of a **dynamic programming** algorithm**.** In general, algorithms which break down a complex problem into subproblems and solve each subproblem optimally are called dynamic programming algorithms.

Learning HMM Model Parameters

Next, you learnt to compute the emission & transition probabilities from a tagged corpus. This process of learning the probabilities from a tagged corpus is called **training an HMM** model. The emission and the transition probabilities can be learnt as follows:

Emission Probability of a word 'w' for tag 't':
P(w|t) = Number of times w has been tagged t/Number of times t appears
Example: P('cat'|N) = Number of times 'cat' appears as Noun/ Number of times Noun is appearing

Transition Probability of tag t1 followed by tag t2:
P(t2|t1) = Number of times t1 is followed by tag t2/ Number of times t1 appears
Example: P(Noun|Adj) = number of times adjective is followed by Noun/ Number of times Adjective is appearing

HMM & Viterbi Implementation in Python

You learnt how to build a POS tagger using Viterbi Heuristic. For training the HMM, i.e., for learning the model parameters, you used the NLTK Treebank corpus. After learning the model parameters, you find the best possible state (tag) sequence for each given sentence. For that, you used the Viterbi algorithm - for every word w in the sentence, a tag t is assigned to w such that it maximises the likelihood of the occurrence of P(tag|word).

P(tag|word) = P(word|tag) * P(tag|previous tag)
= Emission probability * Transition probability

In other words, the tag t is assigned to the word w which has the max P(tag|word).
The assigned tags and words are then stored as a list of tuples. As you move to the next word in the list, each tag to be assigned will use the tag of the previous word.

You saw that the Viterbi algorithm gave ~87% accuracy. The 13% loss of accuracy was majorly because of the fact that when the algorithm hit an unknown word (i.e. not present in the training set), it naively assigned the first tag in the list of tags that we have created.

Deep-learning based POS Tagging

Next, you got a brief overview of how you can build POS taggers using RNNs. **Recurrent Neural Networks (RNNs)** have empirically proven to outperform many conventional sequence models for tasks such as POS tagging, entity recognition, dependency parsing etc. You'll learn RNNs in detail later in the Neural Network course.

Constituency Parsing

Next, you studied why shallow parsing is not sufficient. Shallow parsing, as the name suggests, refers to fairly shallow levels of parsing such as POS tagging, chunking, etc. But such techniques would not be able to check the grammatical structure of the sentence, i.e. whether a sentence is grammatically correct, or understand the dependencies between words in a sentence.

So, you learnt the two most commonly used paradigms of parsing - **constituency parsing and dependency parsing,** which would help to check the grammatical structure of the sentence.

In constituency parsing, you learnt the basic idea of **constituents** as grammatically meaningful groups of words, or phrases, such as noun phrase, verb phrase etc. You also learnt the idea of context-free grammars or CFGs which specify a set of production rules. Any production rule can be written as A -> B C, where A is a **non-terminal** symbol (NP, VP, N etc.) and B and C are either non-terminals or **terminal** symbols (i.e. words in vocabulary such as flight, man etc.). Example a CFG is:

S -> NP VP

NP -> DT N| N| N PP

VP -> V| V NP

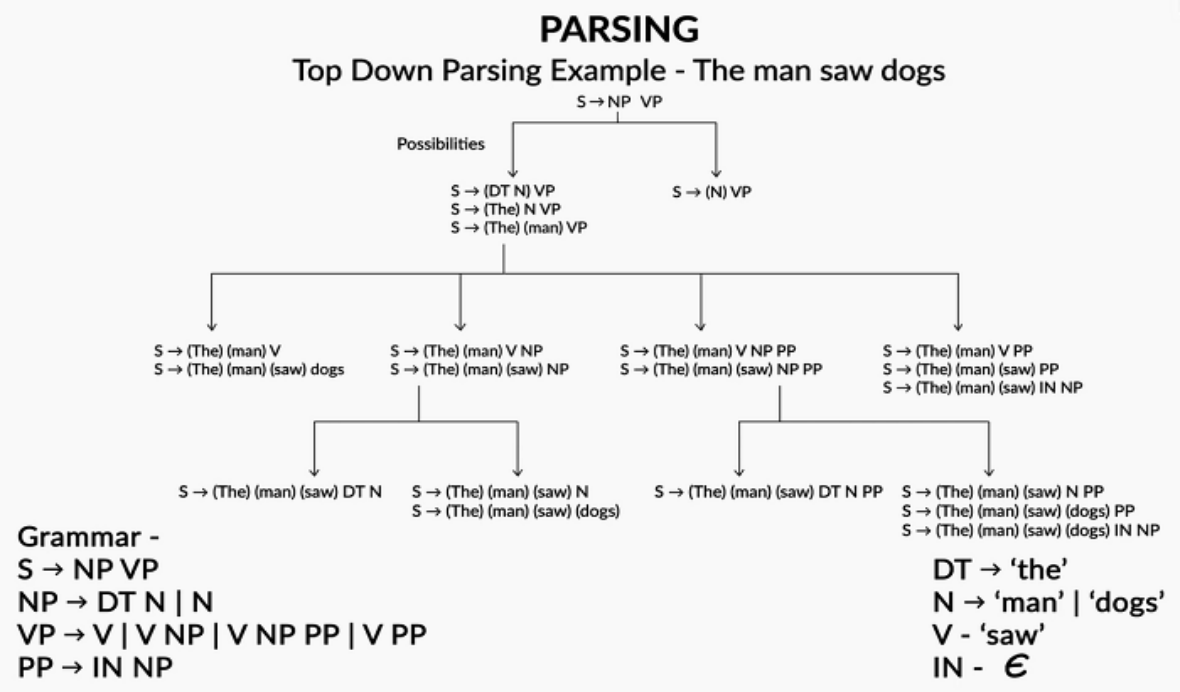
N -> 'man'| 'bear'

V -> 'ate'

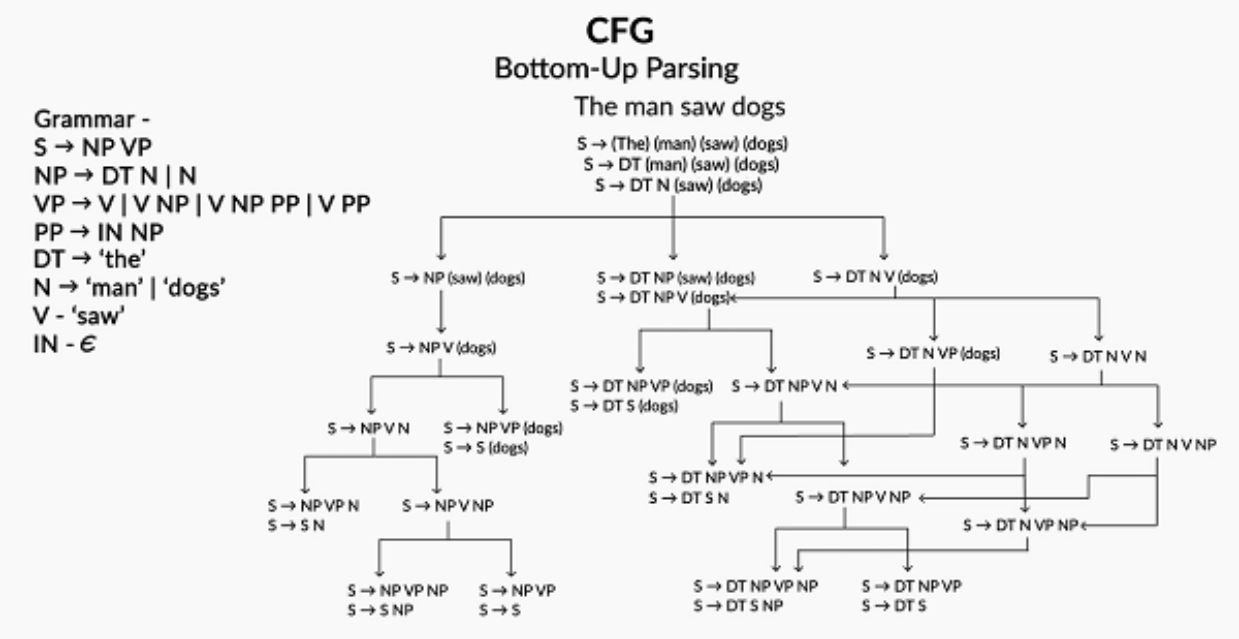
DT -> 'the'| 'a'

Then, you learnt two broad approaches to constituency parsing:

- **Top-down parsing**: starts with the start symbol S at the top and uses the production rules to parse each word one by one. And, you continue to parse until all the words have been allocated to some production rule. Top-down parsers have a specific limitation- **Left Recursion. Example of a left recursion:** *VP -> VP NP.* Whenever a top-down parser encounters such a rule, it runs into an infinite loop, thus no parse tree is obtained. Following is the illustration of top-down parse:



- **Bottom-up parsing:** reduces each terminal word to a production rule, i.e. reduces the right-hand-side of the grammar to the left-hand-side. It continues the reduction process until the entire sentence has been reduced to the start symbol S. You learnt about Shift-Reduce Parser algorithm, which parses the words of the sentence one-by-one either by **shifting** a word to the stack or **reducing** the stack by using the production rules. Below is an example of bottom-up parse tree.



You also learnt how to build both these types of parsed structures in Python.

Probabilistic CFG

Since natural languages are inherently ambiguous (at least for computers to understand), there are often cases where multiple parse trees are possible. In such cases, we need a way to make the algorithms figure out the *most likely* parse tree. **Probabilistic Context-Free Grammars (PCFGs)** are used when we want to find the most probable parsed structure of the sentence. PCFGs are grammar rules, similar to what you have seen before, along with probabilities associated with each production rule. An example production rule is as follows:

NP -> Det N (0.5) | N (0.3) |N PP (0.2)

It means that the probability of an NP breaking down to a 'Det N' is 0.50, to an 'N' is 0.30 and to an 'N PP' is 0.20. Note that the sum of probabilities is 1.00.

Overall probability for a parsed structure of the sentence is probabilities of all rules used in that parsed structure. The parsed tree with maximum probability is best possible interpretation of the sentence. You also learnt to implement PCFG in Python.

Chomsky Normal Form

The **Chomsky Normal Form (CNF)**, proposed by the linguist Noam Chomsky, is a normalized version of the CFG with a standard set of rules defining how production rule must be written. The three forms of CNF rules can be written:

- A -> B C
- A -> a
- S -> ε

A, B, C are non-terminals (POS tags), a is a terminal (term), S is the start symbol of the grammar and ε is the null string. The table below shows some examples for converting CFGs to the CNF:

CFG	VP -> V NP PP	VP -> V
CNF	VP -> V (NP1) NP1 -> NP PP	VP -> V (VP1) VP1 -> ε

Dependency Parsing

After constituency parsing, you learnt about **Dependency Parsing.** In dependency grammar, constituencies (such as NP, VP etc.) do not form the basic elements of grammar, but rather dependencies are established between the words themselves.

You learnt about **free and fixed word order languages**. Free word order languages such as Hindi are difficult to parse using constituency parsing techniques. This is because, in such free-word-order languages, the order of words/constituents may change significantly while keeping the meaning exactly the same. It is thus difficult to fit the sentences into the finite set of production rules that CFGs offer.

Next, you learnt how dependencies in a sentence are defined using the elements Subject-Verb-Object (SVO). The following table shows SVO dependencies in three types of sentences - declarative, interrogative, and imperative:

Declarative	Shyam complimented Suraj Subject Verb Object
Interrogative	Will the teacher take the class today? Aux Subject Object (Aux: auxiliary verbs such as will, be, can)
Imperative	Stop the car! Verb Object

Next, you learnt about universal dependencies. Apart from dependencies defined in the form of subject-verb-object, there's a non-exhaustive list of dependency relationships, which are called universal dependencies.

Dependencies are represented as labelled arcs of the form h → d (l) where 'h' is called the “head” of the dependency, 'd' is the “dependent” and l is the “label” assigned to the arc. In a dependency parse, we start from the **root** of the sentence, which is often a verb. And then start to establish dependencies between root and other words.

Information Extraction

In this session, you learnt to build an **information extraction (IE) system** which can extract entities relevant for booking flights (such as source and destination cities, time, date, budget constraints etc.) in a structured format from unstructured user-generated queries. IE is used in many applications such as **conversational** chatbots, extracting information from encyclopedias (such as Wikipedia), etc. In this session, you learnt to use the ATIS dataset for IE.

A generic IE pipeline is as follows:

1. **Preprocessing**
    1. Sentence Tokenization: sequence segmentation of text.

2. Word Tokenization: breaks down sentences into tokens
3. POS tagging - assigning POS tags to the tokens. The POS tags can be helpful in defining what words could form an entity.

2. **Entity Recognition**
    1. Rule-based models
    2. Probabilistic models

Most IE pipelines start with the usual text preprocessing steps - sentence segmentation, word tokenisation and POS tagging. After preprocessing, the common tasks are **Named Entity Recognition (NER)**, and optionally relation recognition and record linkage. **NER** is arguably the most important and non-trivial task in the pipeline.

You learnt various techniques and models for building **Named Entity Recognition** (NER) system, which is a key component in information extraction systems:

- Rule-based techniques
    - Regular expression based techniques
    - Chunking
- Probabilistic models
    - Unigram & Bigram models
    - Naive Bayes Classifier
    - Decision trees
    - Conditional Random Fields (CRFs)

You learnt about IOB labeling. **IOB (or BIO) method** tags each token in the sentence with one of the three labels: **I - inside (the entity)**, **O- outside (the entity)** and **B - beginning (of entity)**. You saw that IOB labeling is especially helpful if the entities contain multiple words. For example: words like 'Air India', 'New Delhi', etc, are single entities.

Rule-based method for NER

Next, you learnt in detail about Rule-based method: **Chunking.** Chunking is a commonly used shallow parsing technique used to chunk words that constitute some meaningful phrase in the sentence. A **noun phrase chunk** (NP chunk) is commonly used in NER tasks to identify groups of words that correspond to some 'entity'.

Sentence: He bought a new car from the Maruti Suzuki showroom.

Noun phrase chunks - a new car, the Maruti Suzuki showroom

The idea of chunking in the context of entity recognition is simple - most entities are nouns and noun phrases, so rules can be written to extract these noun phrases and hopefully extract a large number of named entities. Example of chunking done using regular expressions:

Sentence: Ram booked the flight.

Noun phrase chunks: 'Ram', 'the flight'

Grammar: 'NP_chunk: {<DT>?<NN>}'

Probabilistic method for NER

Next, you learnt the following two probabilistic models to get the most probable IOB tags for word:

1. **Unigram chunker** computes the unigram probabilities P(IOB label | pos) for each word and assigns the label that is most likely for the POS tag.
1. **Bigram chunker** works similar to a unigram chunker, the only difference being that now the probability of a POS tag having an IOB label is computed using the current and the previous POS tags, i.e. P(label | pos, prev_pos).

Gazetteer Lookup

Another way to identify named entities (like cities and states) is to look up a dictionary or a **gazetteer**. A gazetteer is a geographical directory which stores data regarding the names of geographical entities (cities, states, countries) and some other features related to the geographies.

Machine Learning Classifiers for NER

You studied that just like machine learning classification models, you can have features for sequence labelling task. Features could be the morphology (or shape) of the word such as whether the word is upper/lowercase, POS tags of the words in the neighbourhood, whether the word is present in the gazetteer (i.e. word_is_city, word_is_state), etc. And using these features, you learnt to build a **Naive Bayes classifier and Decision Tree classifier**.

Conditional Random Fields

HMMs can be used for any sequence classification task, such as NER. However, many NER tasks and datasets are far more complex than tasks such as POS tagging, and therefore, more sophisticated sequence models have been developed and widely accepted in the NLP community. One of these models is Conditional Random Fields (CRFs).

CRFs are used in a wide variety of sequence labelling tasks across various domains - POS tagging, speech recognition, NER, and even in computational biology for modelling genetic patterns etc.

Next, you studied the architecture of CRFs. CRFs model the **conditional probability** P(Y|X), where Y is the vector of output sequence (IOB labels here) and X is the input sequence (words to be tagged), which are similar to Logistic Regression classifier. Broadly, there are two types of classifiers in ML:

1. **Discriminative classifiers** learn the boundary between classes by modelling the **conditional probability distribution P(y|x)**, where y is the vector of class labels and x represents the input features. Examples are Logistic Regression, SVMs etc.
1. **Generative classifiers** model the **joint probability distribution P(x,y)**. Examples of generative classifiers are Naive Bayes, HMMs etc.

CRFs are **discriminative probabilistic classifiers** (often represented as undirected graphical models in some texts).

Next, you learnt about CRFs' **feature functions.** CRFs use 'feature functions' rather than the input word sequence x itself. The idea is similar to how features are extracted for building the naive Bayes and decision tree classifiers in a previous section. Some example 'word-features' (each word has these features) are:

- Word and POS tag based features: word_is_city, word_is_digit, pos, previous_pos, etc.
- Label-based features: previous_label

A feature function takes the following **four inputs**:

1. The input sequence of words: ***x***
2. The position of a word in the sentence (whose features are to be extracted)
3. The label y_i of the current word (the target label)
4. The label y_{i-1} of the previous word

Example of a feature function:
A feature function f_1 which returns **1** if the word x_i is a city **and** the corresponding label y_i is 'I-location', else 0. This can be represented as:

$$f_1(x, i, y_i, y_{i-1}) = [[x_i \text{ is in city list name}] \ \& \ [y_i \text{ is I-location}]]$$

The feature function returns 1 only if both the conditions are satisfied, i.e. when the word is a city name and is tagged as 'I-location' (e.g. Chicago/I-location).

Every feature function f_i has a **weight** w_i associated with it, which represents the 'importance' of that feature function. This is almost exactly the same as logistic regression where coefficients of features represent their importance. **Training a CRF** means to **compute the optimal weight vector** w which best represents the observed sequences y for the given word sequences x. In other words, we want to find the set of weights w which maximises P(y|x,w).

In CRFs, the conditional probabilities P(y|x,w) are modeled using a **scoring function**. If there are k feature functions (and thus k weights), for each word i in the sequence x, a scoring function for a word is defined as follows:

$$score_i = exp(w_1.\, f_1 + w_2.\, f_2 \ldots + w_k.\, f_k) = exp(w.\, f(y_i, x_i, y_{i-1}, i))$$

and overall sequence score for the sentence can be defined as:

$$\text{sequence\_score}(y|x) = \prod_{i=1}^{n}(exp(w.\, f(y_i, x_i, y_{i-1}, i))) = exp(\sum_{1}^{n}(w.\, f(y_i, x_i, y_{i-1}, i)))$$

The probability of observing the label sequence **y** given the input sequence **x** is given by:

$$P(y|x, w) = exp(\sum_{1}^{n}(w.\, f(y_i, x_i, y_{i-1}, i)))/Z(x) = exp(w.\, f(x, y))/Z(x)$$

where, Z(x) is sum of scores of all possible tag sequences N:

$$Z(x) = \sum_1^N (exp(w.f(x,y))$$

Training a CRF model means to **compute the optimal set of weights** w which best represents the observed sequences y for the given word sequences x. In other words, we want to find the set of weights w which **maximises the conditional probability** P(y|x,w) for all the observed sequences (x,y):

$$L(w|x,y) = P(Y|X,w) = \prod_1^N (P(y|x))$$

By taking log and simplifying the equations, the final equation comes out as:

$$L(w|x,y) = \sum_1^N (log(exp(w.f(x,y)/Z(x)))$$
$$= \sum_1^N (log(exp(w.f(x,y)) - log(Z(x))) = \sum_1^N (w.f(x,y) - log(Z(x)))$$

To prevent overfitting, we use the regularization term:

$$L(w) = \sum_1^N [(w.f) - log(Z)] \text{ - regularisation\_term}$$

The final equation after taking the **gradient of the log-likelihood function** is:

$$L(w) = \sum_1^N (f(x,y)) - E_{p_v(y|x,w)}(f(x,y')) - 2w/C$$

where $E_{p_v(y|x,w)}(f(x,y')) = \sum_1^N f(x,y') * exp(w.f(x,y'))/z_w(x)$

Prediction using CRF: the inference task to assign the label sequence y· to x which maximises the score of the sequence, i.e.

$$y^* = argmax(w.f(x,y))$$

The naive way to get y^* is by calculating w.f(x,y) for every possible label sequence , and then choose the label sequence that has maximum (w.f(x,y)) value. However, there are an exponential number of possible labels (t^n for a tag set of size t and a sentence of length n), and this task is computationally heavy. You learnt how to derive the best possible path using Viterbi algorithm.

You also learnt the Python implementation of CRF. CRFs outperformed the rule-based and ML-classification algorithms.