Natural Language Processing has always been a key component of Artificial Intelligence (AI). With increase in adoption of AI, systems to automate sophisticated tasks are being built. Some of these examples are described below.

**Diagnosing rare form of cancer**

At the University of Tokyo's Institute of Medical Science, doctors used artificial intelligence to successfully diagnose a rare type of leukemia. The patient was a female in her mid 60s and was initially diagnosed with acute myeloid leukemia. She underwent chemotherapy which attacked her cancer cells. However, her recovery was unusually slow which puzzled the doctors.

The doctors then used an AI system which cross-referenced the patient’s genetic data with tens of millions of oncology papers and diagnosed the cancer as rare secondary leukemia caused by myelodysplastic syndromes. The machine was able to do this in 10 mins, while it would have taken human scientists about 2 weeks to do the same.

**Settling an insurance claim within 3 seconds**

Lemonade built a bot - AI Jim, which interacts with the claimant in real time and understands the nature and severity of the claim. It assesses the likelihood of the claim being fraudulent, and even nudges people to be more honest by incorporating years of behavioral economics research into its conversations. This system settled an insurance claim within 3 seconds by running 18 algorithms.

**Automating customer service tasks**

KLM Royal Dutch Airlines fly to 163 destinations worldwide, operate 200+ aircrafts, and annually ferry 30M+ passengers.

The airline wanted to create “a new entry point” for customers – one that provides opportunities for conversational interactions using voice and text.  They created BB (Blue Bot) – a chat bot that helps customers manage flight bookings though conversational interfaces.

In the first month of launch, the around 1.7 million messages have been exchanged between that bot and 500,000 people.

In all the given examples, the systems are meant to understand the natural language used by human beings. To elaborate, the system that diagnosed cancer had to go through millions of text documents written by humans in english/other languages with a vast vocabulary. Blue Bot not only interprets the queries of a customer which is typically in the natural language, but also generates appropriate responses in natural language.

Having seen examples of where AI systems need to understand and process natural language, let us now understand what is natural language and natural language processing.

# Natural language

In neuropsychology, linguistics, and the philosophy of language, a natural language or ordinary language is any language that has evolved naturally in humans through use and repetition without conscious planning or premeditation [1].

In contrast to artificial languages like Python, C, Java etc. natural languages like English, Portuguese, French etc. have evolved over time & use and it is difficult to express them with strict formal rules.

# Natural Language Processing

Natural Language processing (NLP) is an area of computer science and artificial intelligence concerned with the interactions between computers and human (natural) languages. In particular how to program computers to process and analyze large amounts of natural language data [2].

Let us observe a few sentences.

"**There was not a single man at the party**"  
Does it mean that there were no men at the party? or Does it mean that there was no one at the party?  
Here does 'man' refer to the gender 'man' or 'mankind'?

"**The chicken is ready to eat**"  
Does this mean that the bird (chicken) is ready to feed on some grains? or Does it mean that the meat is cooked well and is ready to be eaten by a human?

"**Google is a great company**" and "**Google this word and find its meaning**"  
Google is being used as a noun in the first statement and as a verb in the second.

"**The man saw a girl with a telescope**"  
Did the man use a telescope to see the girl? or Did the man see a girl who was holding a telescope?

From the above examples, you can see that the natural language is full of ambiguities. Ambiguity can be referred to as the ability to have more than one meaning or being understood in more than one way. This is the primary reason why NLP is considered hard.

Natural languages evolved without a standard rule/logic. They were developed in response to the evolution of human brain and in its ability to understand signs, voice & memory. With NLP, we are now trying to discover rules for something (language) that evolved without rules. These rules can be as simple as using a character 's' to indicate plurality or as difficult as understanding a sarcastic remark. For performing a simple task such as understanding a basic or simple sentence, the machine not only has to understand each word in the sentence but also have to find the connection among the words and figure out the context of the sentence being under consideration.

These are all the factors that make NLP tasks difficult.

Humans communicate through language and this language is represented in form of text. We refer to this text as 'String' or 'combination of Characters' in programming.

Broadly, there are two types of data available which can be in String or Character or Text form. They are Structured and Unstructured data. Structured data can be represented as a table, where there exist rows and columns. Any particular value in structured data can be identified uniquely with a row number and a column number. For example, employee data in a company, where each row represents an employee and each column represents an attribute of an employee (name, address, gender etc).

In case of unstructured data, we do not have data in such a tabular form. It is just in form of plain and simple raw text. For example, tweeter feeds, ticket description of a complaint, text from a book etc.

Note that, even if we are calling these as unstructured data, it still has some structure associated with it. This structure is the grammatical rule that helps us derive meaning to the text. To understand this, let us take the below two sentences into consideration.

**The tiger is sleeping.**

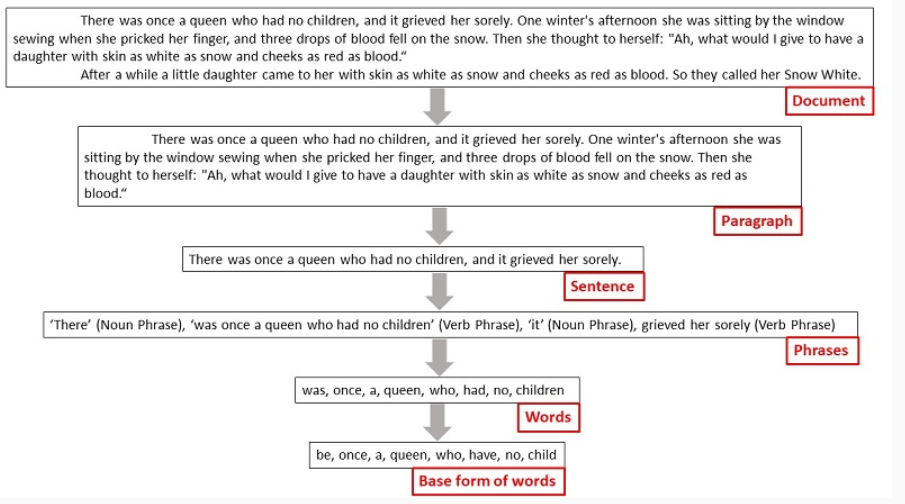
**Sleeping the ? be tiger**

The first sentence clearly makes sense. But considering the second sentence, when we jumble the words or change the position of the punctuation mark or change any word into a different form of itself, the phrase loses its meaning. In Natural Language Processing, we will try to exploit the fact that, a meaningful sentence follows some grammatical rules.

Next, let us understand the hierarchy which is followed by many natural languages in real-life.

A natural language content can be documents or a book or pdf files which has a lot of textual data. On a very high level, we can divide these text into Paragraphs. The Paragraphs can be divided into sentences. The sentences can be further divided into phrases and the phrases can be divided into words. Even some of the words can be converted into its base form.

To understand this, take a look at the below example.



The understanding of this hierarchy will be needed in the later part of this course. Note that, this hierarchy is quite simple for humans to understand. But for the machines to learn this, it actually takes quite a bit of computation. As a matter of fact, we don't have to build every NLP technique from scratch. Since we have rich libraries (like Stanford CoreNLP, Python NLTK) already available which can perform NLP for different kinds of textual data and are customizable as well.

In the subsequent pages, let us discuss the elements of text.

# Elements of Text

Let us now understand some elements of textual data. We shall discuss the following elements of text:

* Tokens
* Vocabulary
* Base of a word
* Part of speech

Note that, we will be working on the elements of text using Stanford CoreNLP in the later part of this course.

# Tokens

Tokens are a meaningful unit of textual data. Depending on the problem we are working on, we may consider every word as a token or every sentence as a token. Most of the time, words are considered as tokens in NLP. The process of breaking the textual data based on the token is called **Tokenization**.

Given data: "India is a republic nation. We are proud Indians."

If we consider every sentence as a token, we will have two tokens from the above data. Usually, punctuation marks are used to perform this kind of tokenization.

'India is a republic nation.', 'We are proud Indians.'

If we consider every word as a token, we will have the below tokens from the above data. Usually, white space characters are used to perform this kind of tokenization.

'India', 'is', 'a', 'republic', 'nation', '.', 'We', 'are', 'proud', 'Indians', '.'

# Vocabulary

The vocabulary of a text is the set of all unique tokens present in it. For example,

Text: "I went to watch a movie. But my friend went to watch another movie."

Here, the vocabulary is: 'I', 'went', 'to', 'watch', 'a', 'movie', '.', 'But', 'my', 'friend', 'another'

# Base of a word

In natural language, the base form of a word is referred to as **Lemma**. For example, the words went, goes, go, gone, going are formed from the base word "go". So "go" is the lemma for the given words. The process of determining the lemma for a given word is called **Lemmatization**.

# Part of speech

Part of speech refers to the category to which a word is assigned based on its function. You may recall that English language has mainly eight parts of speech which are Noun, Verb, Adjective, Adverb, Pronoun, Preposition, Conjunction and Interjection. For example,

Let us take into consideration the following sentence. "**India is a democratic nation.**"

The POS for each word would be: **India/noun is/verb a/determiner democratic/adjective nation/noun ./punctuation**

Now let us understand various components in Natural Language Processing. This will help us derive an approach to tackling any kind of NLP problem. Broadly, there are five components such as:

1. Lexical Analysis
2. Syntactic Analysis
3. Semantic Analysis
4. Discourse Integration
5. Pragmatic Analysis

Let us walk through each one of them.

# Lexical Analysis

Lexicon means the vocabulary of any particular language. Performing any kind of word level or token level analysis on the text corpus is called as Lexical Analysis. For example,

* Dividing a sentence into word tokens
* Identifying the vocabulary in a text
* Finding the base form of the words

Through this kind of analysis we get a feel of what words are predominant in a document. A medical document will contain words specific to medical domain, similarly a banking document will have words specific to banking domain. This kind of analysis can help us perform classification tasks on different documents.

# Syntactic Analysis

Every language has a way of arranging the words in a particular order in the sentence. For example;

"Had boy apple the an." Here, although the words in the sentence seems correct, it is difficult to make sense out of it. The reason being, it is not **Syntactically** correct. We know that English language follows the 'Subject - Verb - Object' structure. According to this rule, the above sentence is not correct. Due to which we have difficulty in understanding the meaning which needs to be conveyed by the sentence. The above sentence should look like below:

"The boy had an apple."

Analysis of the grammatical arrangement of words and studying the relations among them is known as Syntactic Analysis. In the above example, the boy is related to the apple in the sense of having it as food.

Syntactic Analysis is used to extract relations in the text. Note that Syntactic Analysis is dependent on language.

# Semantic Analysis

Semantics deal with the meaning of text. Analyzing the meaningfulness of text is known as Semantic Analysis. The sentence "An apple had the boy." is Syntactically correct, since it agrees with Subject - Verb - Object agreement. But this sentence will be disregarded by Semantic Analysis as an apple cannot eat a boy.

# Discourse Integration

Sentences can either be preceded by some sentences or succeeded by some sentences. To understand the meaning of the current sentence you should know what is the meaning of the previous sentence. The current sentence also helps in understanding the meaning of the subsequent sentence. This kind of analysis is referred to as Discourse Integration.

# Pragmatic Analysis

Pragmatic Analysis deals with understanding the context of any text. For example, Let us consider the sentence "The boy had an apple". Here there can be two possibilities.

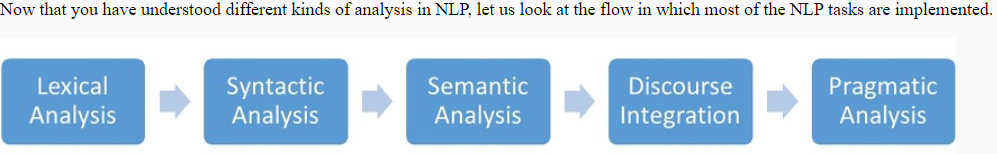
First, perhaps the boy was hungry and he just ate the apple.

Second, perhaps the boy went to a shop, bought an apple and he was just carrying it on the way.

The given sentence can have any of the above meanings. It depends in what context are we speaking. If the sentence would have preceded by some other sentence, then the context would have been set and the meaning would have been clearer.

Let us look at a use case of pragmatic analysis. Say, you have a food chatbot. While having a conversation with this chatbot, you might have different intent at different point of time. You may want to order some food or book a reservation. The chatbot has to identify what intent you are talking about (or what is the context of the conversation), so it can take the necessary action.

Next, let us now look at how to approach an NLP task.



Whenever we have a complex NLP problem, we begin with simple word level analysis (lexical), moving on to the syntax or structure based analysis (syntactic) and then we look at the meaning of the text (semantics). After this we look at the previous and subsequent text (discourse) to identify the context in it (pragmatic). Note that, this complete flow might not be needed all the time. Depending on the use case, you might need one or two components of this flow.

Next, we will define some problems that, we will be working on throughout this course using Stanford CoreNLP.

Problem Statements

Let us look at a few scenarios where NLP can be useful.

# Information Extraction

ABC Consultancy works as a third party company which negotiates merger policies and creates agreements between different firms. They have a lot of agreement documents available. Their company is moving to a digital platform where they have to store these documents along with the metadata like the name of the agreement, the parties involved in the agreement, the start date and the termination date of the agreement, the monetary amount involved in the agreement, the country where the agreement is applicable etc. As there are thousands of such documents, identifying the metadata associated with each of these documents manually is going to cost a lot of time and effort.

The company approached an IT service provider, XYZ Limited to automate this hectic task of extracting information from documents, so that they can save cost and time. Now the job of XYZ Limited is to create some kind of Natural Language Processing application which can automatically extract the relevant information or metadata from every document, thereby providing ABC Consultancy with a great solution.

# Sentiment Analysis

ABCart is an online shopping website where different household products are sold. They want to improve the quality of their products so that it will increase the sale on their website. Therefore, they decide to assess the quality of the sold products. ABCart collects customer reviews for their products which are sold online.

On average, there are hundreds of reviews for each product and thousands of products available online. Finding out which of these products need improvement by manually analyzing all the reviews are going to cost a lot of time and effort.

The company hires an IT service provider, XYZ Limited for this job. XYZ Limited plans to perform Sentiment Analysis using Natural Language Processing on these reviews. This will help them identify the products which have negative, neutral or positive reviews, thereby helping ABCart improve the product quality and increase sale.

Now that we have been introduced to the problem statements, let us learn the tool which will help us solve these.

# Stanford CoreNLP

Stanford CoreNLP is an integrated toolkit/library which provides different functionalities (both pre-built and customizable) to perform Natural Language Processing and is written in Java. This toolkit uses different techniques such as rule-based and probabilistic techniques, deep learning and machine learning. This library is created by the NLP group at Stanford University. The library also gets updated from time to time. CoreNLP is one of the top five NLP tools currently available in the market.

The key features of Stanford CoreNLP include:

* It can perform various operations on raw human language such as finding the base forms of the words, identifying parts of speech of every word in a sentence and many more.
* It is fast, robust, flexible and extensible.
* It provides multiple ways to use Stanford CoreNLP (through command line, Java programmatic API, object oriented simple API, as a simple web service or as third party APIs for some of the common programming languages such as Python).
* It supports multiple human languages (English, Arabic, Chinese, French, German and Spanish)
* Provides pipeline feature for simple execution of NLP functionalities on text

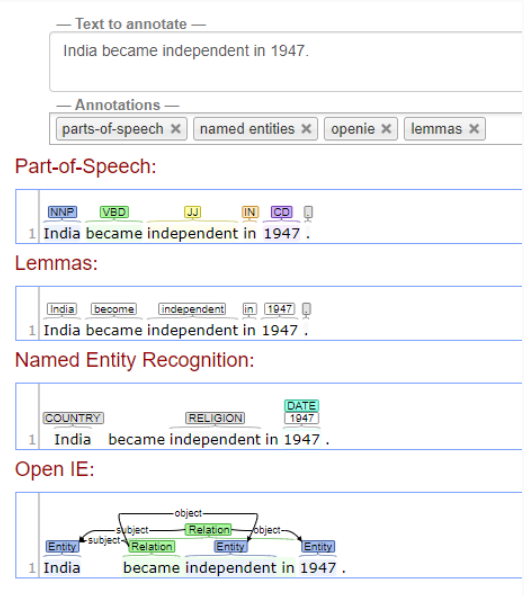
You can check out the official website of Stanford CoreNLP for more details.

The integrated Stanford CoreNLP framework provides multiple Natural Language Processing abilities. These include:

* Identifying parts of speech
* Identifying named entities
* Parsing the sentence to find relationships between parts of a sentence
* Finding the references in a sentence
* Analyzing the sentiment of text
* Extracting binary relations from text

We will deep dive into each one of them in the later part of this course.

We can experiment with some sentences and understand the concepts visually using the official online demo links.

Let us use the following sentence, "**India became independent in 1947.**" Using the official online demo link, we get the following results shown below. Try to understand and interpret these outputs.

Next let us understand what is an Annotation Pipeline in Stanford CoreNLP.

Annotation Pipeline in Stanford CoreNLP

# What is Annotation?

Annotations are something (explained later) that provides some additional information (or metadata) about the text. Stanford CoreNLP package provides some annotators to extract these informations from the text. For example, getting the positions (start and end) of every sentence in a piece of text, getting the POS tags of the words, getting the entities and the type of these entities from the text. Let us observe the below examples.

Given text: "John came to Mexico. He went to Toluca."

Annotated Text for sentence annotation :

sentence 1: start position 0 to end position 20  
sentence 2: start position 21 to end position 39

Annotated Text for entity annotation :

John/**Person** came to Mexico/**Place**. He went to Toluca/**Place**.

Let us put the definitions formally.

**Annotation** and **Annotator** are the two classes that form the backbone of the Stanford CoreNLP package. The results of annotators are stored in the data structure called Annotation. Annotations are basically Maps from keys to bits of the annotation, such as the parse, the POS tags or named entity tags. On the other hand, annotators are like functions, which operates over Annotations instead of Objects. They do things like tokenizing, parsing or finding NER tags in sentences.

Annotators and Annotations are integrated by something called AnnotationPipelines which we will learn in the next few segments.

# Types of Annotators

Stanford CoreNLP has provided many different types of annotators to meet the different information requirements in NLP. Some important annotators are mentioned below.

* **tokenize**: to tokenize the text into words
* **ssplit**:  to split the text and get the sentences
* **pos**:  to obtain the parts of speech of each word token in the sentence
* **lemma**: to get the base form of the words
* **ner**: to get the names of the recognized entities
* **sentiment**: to identify the sentiments in tokens
* **parse**, **depparse**: to parse the text
* **relation**: to extract relation between entities in the text

In this course, we will be going through each one of them in detail. You can find the complete list of annotators on the official website.

# Annotator dependencies

The Annotators are not independent of each other. Let us say, we want to implement sentence splitting (ssplit). For this, we need the tokenized words (tokenize) from the given text. If we want to find the base form of a word (lemma), we need to first tokenize the text (tokenize), then divide it into sentences (ssplit) and find the pos tags for every word (pos). This is called annotator dependency. Some of these dependencies are mentioned below.

| Annotator Dependencies | |
| --- | --- |
| **Annotator Name** | **Required annotators** |
| tokenize | None |
| pos | tokenize, ssplit |
| ner | tokenize, ssplit, pos, lemma |
| sentiment | tokenize, ssplit, pos, parse |
| depparse | tokenize, ssplit, pos |

You can visit the official website to find the complete list of annotator dependencies.

The next question is, how do we take care of annotator dependencies in Stanford CoreNLP? The answer is, we create Annotation Pipeline.

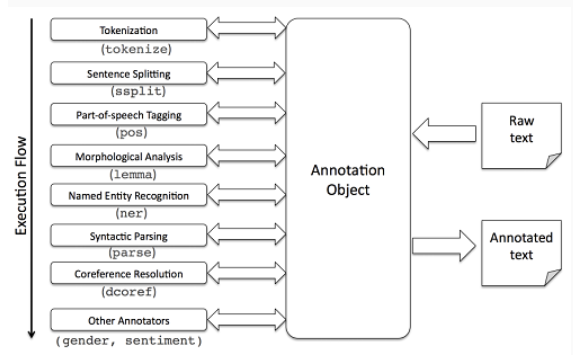
# Annotation Pipeline

An annotation pipeline is a list of annotators to be implemented on text. Annotation pipeline can be used to implement multiple NLP operations on text. It also takes care of annotator dependencies by providing the solution to have multiple annotators.

For example, if we want the sentences in the text and their parts of speech, we can use 'tokenize', 'ssplit' and 'pos' annotators to create the annotation pipeline. The analogy here is that the information from one annotator flows into the other as a fluid in a pipe.

Next, let us learn the architecture of annotation pipeline.

# Architecture of Annotation Pipeline



The raw text is stored in an Annotation object which is basically a Map. Then annotation pipeline (the list of annotators) is run on this object. Some information is extracted by each annotator in this pipeline and the results are written back into the Annotation object. Result of each annotator is stored as different keys in the Annotation object. The execution flow of the annotators is shown in the above diagram. After the pipeline has run, we can extract the annotated text (the results of annotations) from the Annotation object.

Let us now go through the steps on, how we can run the annotators on text.

# Running annotators on the text

The followings are the three basic steps to create Annotators and handle the outputs:

**Step 1**: Create a StanfordCoreNLP object with the necessary annotators. The method **StanfordCoreNLP(Properties props)** creates the pipeline using the annotators given in the '**annotators**' property. You can also set the property for an annotator as shown below.

1. Properties props = new Properties();
2. *// mention the list of annotators in the Properties object that we want to run*
3. props.setProperty("annotators", "tokenize, ssplit, pos, lemma, ner, parse, dcoref");
4. *// set the property for any annotator, here the coref annotator's algorithm is set to use the neural algorithm*
5. props.setProperty("coref.algorithm", "neural");
6. *// build the pipeline*
7. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);

**Step 2**: Run the pipeline on the text. Here we can make use of either of the two classes; **CoreDocument**or **Annotation**. Depending on what class we choose, the way to extract the information would be different.

1. *// create a document object*
2. CoreDocument document = new CoreDocument(text);
3. *// annotate the document*
4. pipeline.annotate(document);
5. *// or...*
6. *// create an empty Annotation*
7. Annotation document = new Annotation(text);
8. *// run all annotators*
9. pipeline.annotate(document);

**Step 3 part 1**: Analysing the CoreDocument output

1. *// get the second sentence as a CoreSentence object*
2. CoreSentence sentence = document.sentences().get(1);
3. *// here we get the list of part of speech tags for 2nd sentence*
4. List<String> posTags = sentence.posTags();
5. System.out.println("Example: pos tags");
6. System.out.println(posTags);

Note that, we will discuss the code blocks in detail in the later part of this course.

**Step 3 part 2**: Analysing the Annotation output

The data structures CoreMap and CoreLabel are used to access the output of annotators. A CoreMap is kind of a Map which uses class objects as keys and the values are of custom types. A CoreLabel is a CoreMap with additional methods and these methods are token specific. The following example shows how we can get the token, pos and named entity data from the text using CoreMap and CoreLabel.

1. *// fetch all the sentences from the Annotation document*
2. List<CoreMap> sentences = document.get(SentencesAnnotation.class);
3. *// traversing the sentences*
4. for(CoreMap sentence: sentences) {
5. *// traversing the words in every sentence*
6. for (CoreLabel token: sentence.get(TokensAnnotation.class)) {
7. *// get the text from the token*
8. String word = token.get(TextAnnotation.class);
9. *// get the POS tag from the token*
10. String pos = token.get(PartOfSpeechAnnotation.class);
11. *// get the NER label from the token*
12. String ne = token.get(NamedEntityTagAnnotation.class);
13. System.out.println("word: " + word + " pos: " + pos + " ne:" + ne);
14. }

Now we have a fair understanding of how Stanford CoreNLP works. Let us now understand some of the major annotators in detail.

Tokenization using Stanford CoreNLP

Tokenization is the process of dividing the text into tokens (here tokens corresponds to words). As the output, the Stanford CoreNLP tokenizer extracts the token, the begin position and the end position of the token. The Stanford CoreNLP tokenizer uses a class called 'PTBTokenizer' (for English language) which provides Unicode compatibility and also supports emojis.

Below are the annotations generated by 'tokenize' annotator. Note that, to access the extracted information from text for all the annotators, the generated annotation classes can be utilized. This is demonstrated on the next page.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| tokenize | TokenizerAnnotator | TokensAnnotation (list of tokens), CharacterOffsetBeginAnnotation, CharacterOffsetEndAnnotation, TextAnnotation (for every token) |

Some of the properties for the 'tokenize' annotator are mentioned below.

| **Property name** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| tokenize.language | Enum {English, German, Spanish, French, Whitespace, Unspecified} | Unspecified | Uses the tokenizer for the mentioned language. By default, it uses the English PTBTokenizer. |
| tokenize.whitespace | boolean | false | If true, the words are separated only when whitespace is present. |
| tokenize.options | String | null | The properties of PTBTokenizer can be specified. |

For more details on Stanford CoreNLP Tokenization and the available options for PTBTokenizer, check out the official website.

Next, let us see how to tokenize given text in Stanford CoreNLP.

Copy the below code, execute it on your system and check the output. Here we have used both CoreDocument and Annotation class. You can use any one of these classes.

1. import java.util.List;
2. import java.util.Properties;
3. import edu.stanford.nlp.ling.CoreAnnotations;
4. import edu.stanford.nlp.ling.CoreLabel;
5. import edu.stanford.nlp.pipeline.Annotation;
6. import edu.stanford.nlp.pipeline.CoreDocument;
7. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
8. public class TokenizeDemo {
9. public static void main(String[] args) {
11. *// set the properties of the pipeline*
12. Properties props = new Properties();
13. props.setProperty("annotators", "tokenize");
15. *// create the pipeline object with the mentioned properties*
16. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
18. *// accessing the tokens in a CoreDocument*
19. CoreDocument document = new CoreDocument("Stanford CoreNLP is a good tool. It is used for multiple NLP applications throughout the world.");
20. pipeline.annotate(document);
22. List<CoreLabel> all\_tokens = document.tokens();
24. *// print all the word tokens and the position information*
25. for (CoreLabel token : all\_tokens) {
26. System.out.println(token.word() + "\t" + token.beginPosition() + "\t" + token.endPosition());
27. }
29. *// accessing the tokens in an Annotation*
30. Annotation document\_annotation = new Annotation("Stanford CoreNLP is a good tool. It is used for multiple NLP applications throughout the world.");
31. pipeline.annotate(document\_annotation);
33. *// getting the first token*
34. *//CoreMap firstToken = exampleAnnotation.get(CoreAnnotations.TokensAnnotation.class).get(0);*
36. *// print all the word tokens and the position information*
37. for (CoreLabel token : document\_annotation.get(CoreAnnotations.TokensAnnotation.class)) {
38. System.out.println(token.word() + "\t" + token.beginPosition() + "\t" + token.endPosition());
39. }
41. }
42. }

At the output, you will get each word, its start position and its end position. Note that the position index starts from 0.

Sample Output:

Stanford    0    8  
CoreNLP    9    16  
is    17    19  
a    20    21

In Stanford CoreNLP, we can divide a text into individual sentences using 'ssplit' annotator. This annotator gives sentences as output by splitting the sequence of tokens into sentences. That means sentence splitting is performed after tokenization. So 'ssplit' annotator is dependent on 'tokenize' annotator.

Note that while providing the annotators in the code, you have to mention it in the order of their execution. So here 'tokenize' is mentioned before 'ssplit'.

Below are the annotations generated by 'ssplit' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| ssplit | WordsToSentencesAnnotator | SentencesAnnotation |

Some of the properties for the 'ssplit' annotator are mentioned below.

| **Property name** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| ssplit.eolonly | Boolean | false | If true, it splits the sentences only at the newline. |
| ssplit.isOneSentence | Boolean | false | If true, each document is treated as one sentence. |
| ssplit.tokenPatternsToDiscard | List(Regex) | null | If provided, this value represents a list of regular expressions for tokens to discard without considering them as boundaries for sentences. |

Check out the official website for more details on Stanford CoreNLP Sentence Split.

Next, let us see how to find the sentences from a given text in Stanford CoreNLP.

Let us go through the below demo. Feel free to experiment with the given code.

1. import java.util.List;
2. import java.util.Properties;
3. import edu.stanford.nlp.ling.CoreAnnotations;
4. import edu.stanford.nlp.pipeline.Annotation;
5. import edu.stanford.nlp.pipeline.CoreDocument;
6. import edu.stanford.nlp.pipeline.CoreSentence;
7. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
8. import edu.stanford.nlp.util.CoreMap;
9. public class SsplitDemo {
10. public static void main(String[] args) {
11. Properties props = new Properties();
12. props.setProperty("annotators", "tokenize, ssplit");
13. *//props.setProperty("ssplit.eolonly", "true");*
14. *// set up pipeline*
15. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
17. *// Accessing the Sentences in a CoreDocument*
18. *// Try adding a \n in the text and set ssplit.eolonly to true*
19. CoreDocument exampleDocument = new CoreDocument("Stanford CoreNLP is a good tool. It is used for multiple NLP application throughout the world. Is it better than other tools?");
20. *// annotate document*
21. pipeline.annotate(exampleDocument);
22. *// access sentences from a CoreDocument*
23. List<CoreSentence> sentences = exampleDocument.sentences();
24. *// this for loop will print out all of the sentences*
25. for (CoreSentence sent : sentences) {
26. System.out.println(sent);
27. }
29. *// Accessing the Sentences in an Annotation*
30. Annotation exampleAnnotation = new Annotation("Stanford CoreNLP is a good tool. It is used for multiple NLP application throughout the world. Is it better than other tools?");
31. pipeline.annotate(exampleAnnotation);
32. List<CoreMap> sentences1 = exampleAnnotation.get(CoreAnnotations.SentencesAnnotation.class);
33. *// print all the sentences*
34. for (CoreMap sent : sentences1) {
35. System.out.println(sent);
36. }
38. }
39. }

After execution, you should get the below output.

Stanford CoreNLP is a good tool.  
It is used for multiple NLP application throughout the world.  
Is it better than other tools?

# Part-of-speech tagging

We are already familiar with what parts of speech are. In natural language, there are some ambiguities which can be resolved through parts of speech. To understand this, let us observe the below two sentences.

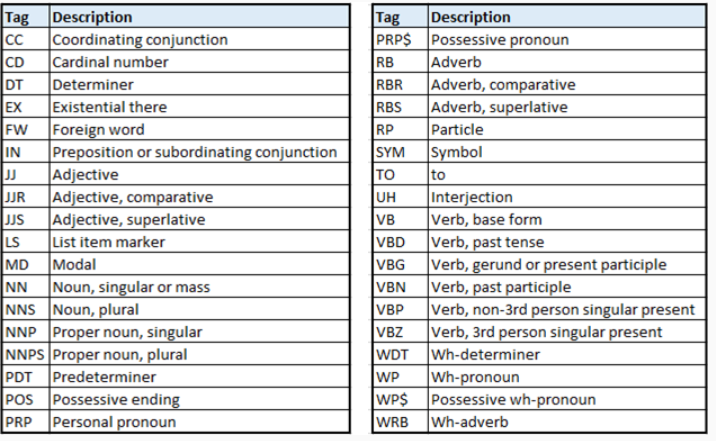
You should never break a promise (noun).  
We promise (verb), we will pray every morning.

As you can see, both the words are same, but one of them is used as a noun while the other represents an action or verb. To understand the meaning in a natural language (syntactic or semantic), it is mandatory that we find the parts of speech of every word. The parts of speech may not directly give the meaning to any natural language, but it definitely helps us find it.

Part-of-speech tagging (POS tagging) is also called as 'grammatical tagging' or 'word-category disambiguation'. It is the process of marking the words in a text or corpus with its corresponding part of speech. This process is based on both the definition or syntax of the word (achieve, achievement) and its context. Context means, the relationship of a word with its adjacent and related words the given phrase or sentence. As shown in the above example, same words (promise) can have different pos based on its context.

In NLP, we represent the parts of speech with tags. Noun is represented as NN, Verb as VB, Adjective as JJ, Adverb as RB, Pronoun as PR, Preposition as IN, Conjunction as CC, Interjection as UH etc. In computational applications, more fine-grained POS tags are also used. For example, NNS represents plural noun, NNP represents proper noun, VBD represents past verb etc.

Note that, different languages may have different POS tagset. For English language, the most popular pos tagset used for NLP (also in Stanford CoreNLP) is the Penn Treebank tagset. It has 36 different tags for English words and some tags for punctuation marks. The tagset for English words in Penn Treebank are shown below.



Let's observe the following sentence and the pos tags of the respective words. "**India is a democratic nation.**"

POS tags:**India/NNP is/VBZ a/DT democratic/JJ nation/NN ./.**

Another example:**Karma/NN of/IN humans/NNS is/VBZ AI/NNP ./.**

Stanford CoreNLP provides us with annotator 'pos' which assigns pos tags to every token or word in the text. At the backend, it uses a log-linear pos tagger which has been updated from time to time to increase its speed, performance, usability and support for other languages. Note that, 'pos' annotator is dependent on both 'tokenize' and 'ssplit' annotator.

Below are the annotations generated by 'pos' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| pos | POSTaggerAnnotator | PartOfSpeechAnnotation |

The properties for the 'pos' annotator are mentioned below.

| **Property Name** | **Description** |
| --- | --- |
| pos.model | By default, english left3words POS model is used. It can be changed to use any other POS model. |
| pos.maxlen | This is the maximum sentence size to be used for POS tagging. It is useful for noisy text, which doesn't have punctuation marks. |

For POS tagging, Stanford CoreNLP provides you with multiple trained models for english language and trained models for Arabic, Chinese, French, Spanish and German languages. It also provides you with a framework to create or train your own custom pos tagger in any language. Refer to the official Stanford NLP website for more details.

Next, let us see how to find the pos tags from the given text using Stanford CoreNLP.

Let us look at the below demo. Here, first we are annotating the document, then for each sentence, we are extracting the words and their pos tags. Look at the output and try to verify if it is correct or not.

1. import java.util.List;
2. import java.util.Properties;
3. import edu.stanford.nlp.ling.CoreLabel;
4. import edu.stanford.nlp.ling.CoreAnnotations.PartOfSpeechAnnotation;
5. import edu.stanford.nlp.ling.CoreAnnotations.SentencesAnnotation;
6. import edu.stanford.nlp.ling.CoreAnnotations.TextAnnotation;
7. import edu.stanford.nlp.ling.CoreAnnotations.TokensAnnotation;
8. import edu.stanford.nlp.pipeline.Annotation;
9. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
10. import edu.stanford.nlp.util.CoreMap;
11. public class PosDemo {
12. public static void main(String[] args) {
13. String text = "Joe Smith was born in California. In 2017, he went to Paris, France in the summer.";
15. Properties props = new Properties();
16. *// provide the annotators to run*
17. props.setProperty("annotators", "tokenize, ssplit, pos");
19. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
20. *// create an Annotation document object*
21. Annotation document = new Annotation(text);
23. pipeline.annotate(document);
25. List<CoreMap> sentences = document.get(SentencesAnnotation.class);
27. for(CoreMap sentence: sentences) {
28. *// traversing the words in the current sentence*
29. for (CoreLabel token: sentence.get(TokensAnnotation.class)) {
30. *// this is the text of the token*
31. String word = token.get(TextAnnotation.class);
32. *// this is the POS tag of the token*
33. String pos = token.get(PartOfSpeechAnnotation.class);
35. System.out.println(String.format("word: [%s] pos: [%s]", word, pos));
36. }
37. }
38. }
39. }

After execution, you should get the below output.

word: [Joe] pos: [NNP]  
word: [Smith] pos: [NNP]  
word: [was] pos: [VBD]  
word: [born] pos: [VBN]  
word: [in] pos: [IN]  
word: [California] pos: [NNP]  
word: [.] pos: [.]  
word: [In] pos: [IN]  
word: [2017] pos: [CD]  
word: [,] pos: [,]  
word: [he] pos: [PRP]  
word: [went] pos: [VBD]  
word: [to] pos: [TO]  
word: [Paris] pos: [NNP]  
word: [,] pos: [,]  
word: [France] pos: [NNP]  
word: [in] pos: [IN]  
word: [the] pos: [DT]  
word: [summer] pos: [NN]  
word: [.] pos: [.]

Lemmatization

Lemmatization is the process of finding the base form (Lemma) of a word. It is used to find a unique set of vocabulary for a document and perform morphological analysis of words. A word can take many different forms, for example, democracy, democratic and democratization. Consider a document which has all such words. If we consider each and every word individually, it might increase the computational cost for text analysis. Therefore sometimes, instead of using all the derived forms of a word, we can simply replace them with the base form. Lemmatization can be useful when building applications like search engine.

In Stanford CoreNLP, to perform Lemmatization, we can use 'lemma' annotator. This annotator generates the lemmas of the words for every token in the text corpus. It is dependent on tokenize, ssplit and pos annotators. Below are the annotations generated by 'lemma' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| lemma | MorphaAnnotator | LemmaAnnotation |

Next, let us go through a demo on how to use 'lemma' in Stanford CoreNLP.

1. import java.util.List;
2. import java.util.Properties;
3. import edu.stanford.nlp.ling.CoreLabel;
4. import edu.stanford.nlp.ling.CoreAnnotations.LemmaAnnotation;
5. import edu.stanford.nlp.ling.CoreAnnotations.PartOfSpeechAnnotation;
6. import edu.stanford.nlp.ling.CoreAnnotations.SentencesAnnotation;
7. import edu.stanford.nlp.ling.CoreAnnotations.TextAnnotation;
8. import edu.stanford.nlp.ling.CoreAnnotations.TokensAnnotation;
9. import edu.stanford.nlp.pipeline.Annotation;
10. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
11. import edu.stanford.nlp.util.CoreMap;
12. public class LemmaDemo {
13. public static void main(String[] args) {
14. String text = "Veer had gone for shopping. He would be coming back in 30 minutes.";
16. Properties props = new Properties();
18. props.setProperty("annotators", "tokenize, ssplit, pos, lemma");
20. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
22. Annotation document = new Annotation(text);
24. pipeline.annotate(document);
26. List<CoreMap> sentences = document.get(SentencesAnnotation.class);
28. for(CoreMap sentence: sentences) {
30. for (CoreLabel token: sentence.get(TokensAnnotation.class)) {
32. String word = token.get(TextAnnotation.class);
34. String lemma = token.get(LemmaAnnotation.class);
36. System.out.println(String.format("word: [%s] lemma: [%s]", word, lemma));
37. }
38. }
39. }
40. }

After execution, you should get the below output.

word: [Veer] lemma: [veer]  
word: [had] lemma: [have]  
word: [gone] lemma: [go]  
word: [for] lemma: [for]  
word: [shopping] lemma: [shopping]  
word: [.] lemma: [.]  
word: [He] lemma: [he]  
word: [would] lemma: [would]  
word: [be] lemma: [be]  
word: [coming] lemma: [come]  
word: [back] lemma: [back]  
word: [in] lemma: [in]  
word: [30] lemma: [30]  
word: [minutes] lemma: [minute]  
word: [.] lemma: [.]

## **Named Entity Recognition**

Let us consider the below sentences.

'John wants to order a medium size nonveg pizza with extra cheese topping.'  
'Please recharge a 100 rupees topup on 9876543210.'  
'What are the available flights on 22nd May 2019 from Bangalore to Delhi?'

In all the sentences, there is some information which needs to be extracted for the respective business purpose. These extracted pieces of information also need to be classified into a proper category. For example,

'John (PERSON) wants to order a medium (PIZZA\_SIZE) size nonveg (PIZZA\_TYPE) pizza with extra cheese (TOPPING) topping.'  
'Please recharge a 100 (RECHARGE\_AMOUNT) rupees topup on 9876543210 (MOBILE\_NO).'  
'What are the available flights on 22nd May 2019 (DATE) from Bangalore (FROM\_LOCATION) to Delhi (TO\_LOCATION)?'

The process of extracting and identifying named entities (PERSON, LOCATION, DATE etc) form textual data is known as Named Entity Recognition (NER).

NER has multiple numbers of industry applications, such as conversation engines, application auto filling, document classification etc. Stanford CoreNLP provides functionalities for NER tagging. Let us learn the 'ner' annotation.

In Stanford CoreNLP, we can use 'ner' annotator to perform Named Entity Recognition. This annotator is dependent on tokenize, ssplit, pos and lemma annotators.

The 'ner' annotator recognizes numerical (NUMBER, PERCENT, MONEY, ORDINAL), temporal (TIME, DATE, SET, DURATION) and named (ORGANIZATION, LOCATION, PERSON, MISC) entities (in total 2 classes). If we use 'regexner'\* annotator along with fine-grained entity recognition, 11 more classes are supported which are URL, EMAIL, NATIONALITY, STATE\_OR\_PROVINCE, CITY, COUNTRY, CAUSE\_OF\_DEATH, (job) TITLE, RELIGION, CRIMINAL\_CHARGE, IDEOLOGY. This makes the annotator recognize 23 classes in total.

'ner' annotator uses both machine learning based models (CRF or Conditional Random Fields) especially for named entities and rule-based systems for numerical or temporal entities for tagging. To recognize the named entities, three CRF sequence taggers are used in combination which are trained on different corpora. These three models include a 3 class model (Organization, Person, Location), a 4 class model (Location, Person, Organization, Misc) and a 7 class model (Location, Person, Organization, Money, Percent, Date, Time). The default 'ner' annotator uses several entity recognizers (crf models or rule-based systems) at the back end and then combines their results. But it can also run a single model or only a rule-based system as well.

The complete named entity recognition pipeline has become somewhat complex which involves multiple distinct phases integrated with each other. You can go through the official documentation to understand these distinct phases in more detail.

Below are the annotations generated by 'ner' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| ner | NERCombinerAnnotator | NamedEntityTagAnnotation, NormalizedNamedEntityTagAnnotation |

Some of the properties for the 'ner' annotator are mentioned below.

| **Property name** | **Type** | **Default Value** | **Description** |
| --- | --- | --- | --- |
| ner.model | List(String) | null | A comma-separated list of NER model name(s) can be provided. By default, a list of three English models is used. The provided names will be searched for as filenames, classpath resources or URLs. |
| ner.applyNumericClassifiers | boolean | true | To use the numeric classifiers or not which includes the SUTime\*. Only valid for English language, so if using a different language, set the option to false. |
| ner.applyFineGrained | boolean | true | To apply fine-grained NER tags or not. This slows down the performance. |

You can visit the Stanford CoreNLP official website for more details on various properties of 'ner' annotator. Also, Stanford CoreNLP supports training of customized NER models. You can also download models for different languages like German, Spanish and Chinese.

Next, let us understand how to use the 'ner' annotator in Stanford CoreNLP with a demo.

\*'regexner' annotator is a sub-annotator of 'ner' annotator which provides and implements a rule-based, simple NER system over sequences of tokens using an extension of Java regexes.

\*SUTime is a library to recognize and normalize the time expressions. For example, 'next Monday at 4 pm' will be converted to something like '2019-06-24T16:00' depending on the current reference time.

This demo prints the detected entities and their entity types in the given text. Feel free to experiment with the given code.

1. import edu.stanford.nlp.ling.CoreLabel;
2. import edu.stanford.nlp.pipeline.\*;
3. import java.util.List;
4. import java.util.Properties;
5. public class NerDemo {
6. public static void main(String[] args) {
8. *// set up pipeline properties*
9. Properties props = new Properties();
10. props.setProperty("annotators", "tokenize,ssplit,pos,lemma, ner");
12. *// try disabling the fine-grained ner and check the output*
13. *//props.setProperty("ner.applyFineGrained", "false");*
14. *// set up pipeline*
15. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
16. *// create an document object*
17. CoreDocument doc = new CoreDocument("Birupakshya Mahapatra is travelling to Bangalore for 3 days.");
19. *// annotate the created document object*
20. pipeline.annotate(doc);
22. *// view results*
23. System.out.println("---");
24. System.out.println("entities found");
25. for (CoreEntityMention em : doc.entityMentions())
26. System.out.println("\tdetected entity: \t"+em.text()+"\t"+em.entityType());
27. System.out.println("---");
28. System.out.println("tokens \t ner tags");
29. List<CoreLabel> tokensAndNERTags = doc.tokens();
30. for (CoreLabel token : tokensAndNERTags) {
31. System.out.println(token.word() + "\t" + token.ner());
32. }
33. }
34. }

After execution, you should get the below output.

---  
entities found  
    detected entity:     Birupakshya Mahapatra    PERSON  
    detected entity:     Bangalore    CITY  
    detected entity:     3 days    DURATION  
---  
tokens      ner tags  
Birupakshya    PERSON  
Mahapatra    PERSON  
is    O  
travelling O  
to    O  
Bangalore    CITY  
for    O  
3    DURATION  
days    DURATION  
.    O

# Regular Expressions in Java

A regular expression (regex) is a special text or string which describes a particular pattern for search operations on text. For any NLP practitioner regex is a very essential tool. Regular expressions can be used to search, edit, or manipulate textual data.

Stanford NLP regular expression classes are very similar to the classes provided in core Java. Let us begin with a small primer on Java regular expressions.

Some examples where regexes can be used are: We want to search a list of sentences to find which the sentences contain the word 'India', which sentences contain numbers, which sentences start with the word 'Infosys' etc.

Java provides the java.util.regex package for implementing regular expressions. This package has three primary class, which are:

* Pattern Class: representation of a regular expression or the pattern that we want to search
* Matcher Class: matches the specified pattern in the given string
* PatternSyntaxException: handles the errors

Let us look at the below demo which extracts a 10 digit phone number from the given String.

1. import java.util.regex.\*;
2. public class JavaRegexDemo {
3. public static void main(String[] args) {
5. String line = "My phone number is 9876543210";
6. String pattern = "(\\d{10})";
7. *// Create a Pattern object*
8. Pattern r = Pattern.compile(pattern);
9. *// Create a matcher object.*
10. Matcher m = r.matcher(line);
11. if (m.find( )) {
12. System.out.println("Found value: " + m.group(0));
14. }else {
15. System.out.println("NO MATCH");
16. }
17. }
18. }

Output:

Found value: 9876543210

Now that we are familiar with the basic Java regex, let us understand the Stanford NLP tokens regex.

Stanford CoreNLP provides a library called TokensRegex which is a powerful and complex framework used for identifying patterns over text and tokens. This is achieved through the annotator 'tokensregex'. The annotator class name generated by this annotator is called 'TokensRegexAnnotator'.

TokensRegex is used to create rule-based systems to search patterns and perform actions when patterns are detected. We can do this in two ways; we can either build a TokensRegex pipeline (which is used with StanfordCoreNLP pipeline) or we can run it as an annotator. Here, we are going to focus on the later part. You can refer to the Stanford CoreNLP official website for more information on the former part. Both of these ways require some specific rules to be defined for pattern recognition. We will learn how to write these rules in detail in a while.

When we want to find or identify some pattern, what kind of pattern are we talking about? How identifying these patterns with tokensregex is different from using simple java regex. Let us look into an example. Consider the below sentences.

'I want to book a flight from Bangalore to Delhi', 'Can you book a ticket to Pune starting from Chennai', 'Is there any available flight going from Hyderabad to Bangalore'

Let us say, we want to create a rule-based information extraction system. If we observe the above three sentences, we can see that there is a common pattern present in all of them. The pattern is, there are two locations present in all the sentences. If we observe closely, we can see that, the location which is followed by 'from' should be the 'departure\_location' and the location which is followed 'to' should be the 'destination\_location'. Identifying these kinds of patterns (first location tag or pos tag, then a combination of word and these tags) would not have been possible with simple Java regexes. But using Stanford NLP, we can create such rules and implement them on text to identify the patterns.

We can create any such rule-based system using TokensRegex. For example, identifying the date from a given expression. One such temporal rule-based tagger in Stanford NLP is 'SUTime' which has been developed using TokensRegex.

Let us now examine one such rules file and understand how to write them.

Below is a rules file ('[basic\_ner.rules](https://lex.infosysapps.com/content-store/Infosys/Infosys_Ltd/Public/lex_auth_012771566666162176959/web-hosted/assets/basic_ner.rules" \t "_blank)') that implements a TokensRegex pipeline.

1. *# make all patterns case-sensitive*
2. ENV.defaultStringMatchFlags = 0
3. ENV.defaultStringPatternFlags = 0
4. *# the below mentioned Java classes shall be used by the written rules*
5. ner = { type: "CLASS", value: "edu.stanford.nlp.ling.CoreAnnotations$NamedEntityTagAnnotation" }
6. tokens = { type: "CLASS", value: "edu.stanford.nlp.ling.CoreAnnotations$TokensAnnotation" }
7. *# define some regexes over tokens*
8. $COMPANY\_BEGINNING = "/[A-Z][A-Za-z]+/"
9. $COMPANY\_ENDING = "/(Corp|Inc)\.?/"
10. *# rules for identifying names of companies*
11. { ruleType: "tokens", pattern: ([{word:$COMPANY\_BEGINNING} & {tag:"NNP"}]+ [{word:$COMPANY\_ENDING}]), action: Annotate($0, ner, "COMPANY"), result: "COMPANY\_RESULT" }

The first section affects the environment of the TokensRegex pipeline. By setting these values to 0, we are making the rules case-sensitive. You can check the official documentation for other environment class (edu.stanford.nlp.ling.tokensregex.Env) variables for TokensRegex pipeline.

The second section deals with binding certain variables to Java classes which are used as annotation keys. Here, we bind “ner” and “tokens” to the respective annotation keys.

In the third section, we bind some variables to regular expressions. Here, the $COMPANY\_BEGINING matches with those tokens which start with a capital letter and has only letters in it, for example, 'Infosys'. The $COMPANY\_ENDING matches with tokens like 'Corp', 'Inc', 'Corp.' and 'Inc.'. By defining these variables, we need not write regexes multiple times and we can also define large regex patterns. Note that, for exact string match, we can write the expression directly within "", but for regular expression match we also have to write the patterns within //.

The next section defines the rule to match the token pattern. The 'ruleType' of the given rule is 'tokens', which means we want to identify patterns over sequences of tokens. For other rule types check out the official documentation.

The “pattern” of the rule defines the actual pattern we want to see in the tokens. In our case, we want to see $COMPANY\_BEGINNING that have the “NNP” pos tag and ends with a token that matches the $COMPANY\_ENDING pattern. Note that, there is () around the pattern which indicates a capture group.

If the pattern gets matched, then the 'action' part of the rule gets executed. Typically, this means we want to annotate all of the tokens in the matched pattern in some manner. This is achieved with the help of the 'Annotate()' function. Here, we are stating that annotate all the matched tokens in the pattern (which is indicated by the group $0 in the token pattern) and set their 'CoreAnnotation.NamedEntityTagAnnotation.class' value (indicated by 'ner') to the value 'COMPANY'. This is where the actual CoreLabel’s are being altered by having their CoreAnnotations.NamedEntityTagAnnotation.class field changed. You can go through the official documentation for other available action functions.

Finally, we may save the result as a value for property 'tokensregex.matchedExpressionsAnnotationKey' which can be used in the Java code. So we can make the rule return a 'result' whenever it fires. Here, the value of the matched expression will be set to 'COMPANY\_RESULT'.

Now that we have understood a basic rules file for entity tagging, let us learn some more syntaxes which will help us write proper rules.

Below are the standard names for the annotation keys which are frequently used. There are many others as well.

| **Name** | **Annotation Class** |
| --- | --- |
| word | CoreAnnotations.TextAnnotation |
| tag | CoreAnnotations.PartOfSpeechTagAnnotation |
| lemma | CoreAnnotations.LemmaAnnotation |
| ner | CoreAnnotations.NamedEntityTagAnnotation |
| normalized | CoreAnnotations.NormalizedNamedEntityTagAnnotation |

Below are the summary of bracketing symbols which are used in TokenRegex.

| **Symbol** | **Description** |
| --- | --- |
| [ . . . ] | Indicate one token |
| ( . . . ) | Indicate grouping (capture groups) |
| { . . . } | Indicate the frequency of a group to be repeated or the attributes which match an expression |
| / . . . / | Indicate regular expression patterns |
| " . . . " | Indicate a string |

Let us look at the below expressions.

[ { word:"cat" } ] matches the token which has text "cat"  
[ { word:/cat|dog/; tag:"NN" } ] matches a token with text "cat" or "dog" and NN as the pos tag  
[ { word<=8 } ] matches the token with text which has numeric value less than or equal to 8, we can also use ==, !=, < etc.

Now, try to interpret the below expression.  
([{ner: PERSON}]+) /was|is/ /an?/ []{0,3} /painter|artist/

Note that we can also use | (or), & (and), ! (negation) operations to create compound expressions like:  
[ {word>=2000} & {word <=3000} ] matches if token has text which is a number between 2000 and 3000  
[ {word::IS\_NUM} | {tag:CD} ] matches if the token text has pos tag CD or it is numeric

Let us look at the below demo on how to use tokensregex. Note that, you have to provide the rules file for executing the code. The rules file which is explained earlier is used here. Try to experiment with the code, write your own rules and interpret the output.

1. import edu.stanford.nlp.ling.\*;
2. import edu.stanford.nlp.pipeline.\*;
3. import edu.stanford.nlp.util.\*;
4. import java.util.Properties;
5. public class TokenRegexDemo {
6. public static void main(String[] args) {
8. Properties props = new Properties();
9. props.setProperty("annotators", "tokenize,ssplit,pos,lemma, tokensregex");
10. props.setProperty("tokensregex.rules", "src/main/java/corenlp/basic\_ner.rules" );
11. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
13. Annotation ann = new Annotation("There will be a big announcement by Apple Inc today at 5:00pm. She has worked at Miller Corp. for 5 years. He works for apple inc in cupertino.");
14. pipeline.annotate(ann);
15. *// show results*
16. System.out.println("tokens\n");
17. for (CoreMap sentence : ann.get(CoreAnnotations.SentencesAnnotation.class)) {
18. for (CoreLabel token : sentence.get(CoreAnnotations.TokensAnnotation.class)) {
19. System.out.println(token.word() + "\t" + token.ner());
20. }
21. System.out.println("---");
22. }
23. }
24. }

After executing the above code, you should get the following output:

tokens

There    null  
will    null  
be    null  
a    null  
big    null  
announcement    null  
by    null  
Apple    COMPANY  
Inc    COMPANY  
today    null  
at    null  
5:00    null  
pm    null  
.    null  
---  
She    null  
has    null  
worked    null  
at    null  
Miller    COMPANY  
Corp.    COMPANY  
for    null  
5    null  
years    null  
.    null  
---  
He    null  
works    null  
for    null  
apple    null  
inc    null  
in    null  
cupertino    null  
.    null

Introduction to Parsing

Parsing is a type of Syntactic Analysis which divides the sentence or text into its constituent parts and identifies the dependencies among them. Alternatively, we can say, Parsing is a technique which checks the sentence based on the underlying grammatical structure. Let us consider the below example.

'the cat is playing with the black dog'

An NLP parser should be able to successfully identify the noun phrases ('the cat', 'the black dog'), the verb phrases ('is playing with the black dog') and any other phrases from the sentence as well as the relations among words like dog -> black or playing -> cat etc.

There are two types of Parsing techniques.

* Constituency Parsing
* Dependency Parsing

In the next few segments, we will discuss each one of them with detail. Let us learn about some of the applications where Parsing can be used.

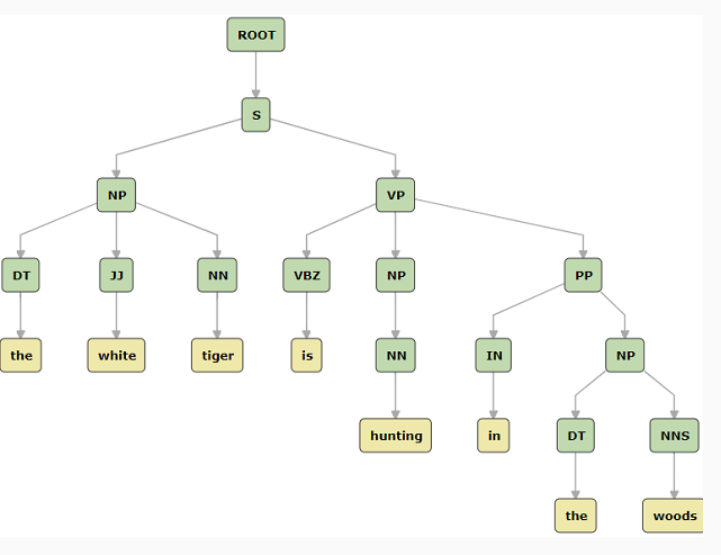
**Why do we need Parsing?** To answer this, let us look at the below applications.

* Information Extraction: Using Parsing we can extract the key phrases from the text and relation among them.
* Text Summarization: Shortening a large document to only important phrases, which gives the summary of the entire document.
* Question Answering System: After identifying the relations among the parts of the sentence, we can easily answer any question about the sentence. For example, 'the cat is playing with the dog'. The parser has identified the relation playing > subject > cat. Now, if you ask a question like 'Who is playing?' The answer would be 'the cat'.
* Machine Translation: Different languages have different grammatical structures or word orders. But the relations among the words are universal across languages. For example, some language might follow the SVO rule (Subject Verb Object) and some might follow SOV rule.
* Grammar Checking: To check if a sentence is syntactically correct or not.
* Text to Speech Synthesis: Converting a written text into speech.

Since you are now familiar with some of the applications, let us look at various types of parsing and implement them using Stanford CoreNLP.

Constituency Parsing

Constituency Parsing is the technique to divide a sentence into its constituent parts or phrases. Let's consider the sentence, 'the white tiger is hunting in the woods'. The Constituency Parsing will result in a tree structure shown below.

The above tree can also be represented as below:

(ROOT (S (NP (DT the) (JJ white) (NN tiger)) (VP (VBZ is) (VP (VBG hunting) (PP (IN in) (NP (DT the) (NNS woods)))))))

Let us understand this structure. The root or the starting point here is the sentence (S). The sentence is divided into two parts, Noun Phrase (NP which is 'the white tiger') and Verb phrase (VP which is 'is hunting in the woods').

The NP is further divided into three parts (determinant, adjective, noun). Similarly, the VP is divided into a verb, NP and a prepositional phrase (PP). This division continues until we reach the exact words present in the sentence.

Note that, arriving at a tree structure from a sentence is done by trained probabilistic models at the back end. In this course, we are not going to discuss on how to create those models, rather we will learn how to parse a sentence using Stanford CoreNLP.

Let us now discuss the 'parse' annotator provided by Stanford CoreNLP.

In Stanford CoreNLP, we can perform constituency parsing using 'parse' annotator. This annotator is also be used for dependency parsing, but if we are performing only dependency parsing, we should use the 'depparse' annotator as it is much faster. 'parse' annotator is dependent on tokenize and ssplit annotators.

Below are the annotations generated by 'parse' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| parse | ParserAnnotator | TreeAnnotation, BasicDependenciesAnnotation, EnhancedDependenciesAnnotation, EnhancedPlusPlusDependenciesAnnotation |

The constituent based outputs are saved in the TreeAnnotation and the dependency based outputs are saved in the other three annotations shown above.

Some of the properties for the 'parse' annotator are mentioned below.

| **Property name** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| parse.model | file, classpath resource or URL | edu/stanford/nlp/models/lexparser/englishPCFG.ser.gz | This option can be used to set a different model for parsing. |
| parse.maxlen | integer | -1 | Maximum length of a sentence (as no. of tokens) to be parsed. if set to a positive integer. Useful, when parsing long sequences. |

You can go through the official website for more details on various options.

There are multiple trained models available for parsing. One such model is a shift-reduce constituency parser which is faster and more memory efficient, but it takes some time to load. You can download the model from the official website. We can also train our own shift-reduce parser. Note that for shift-reduce parser, we have to specify 'pos' annotator externally.

Next, let us see how to implement constituency parsing in Stanford CoreNLP.

Let us look at the below demo. You can try using the shift-reduce parser.

1. import edu.stanford.nlp.ling.CoreAnnotations;
2. import edu.stanford.nlp.pipeline.Annotation;
3. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
4. import edu.stanford.nlp.trees.\*;
5. import java.util.\*;
6. public class ParseDemo {
7. public static void main(String[] args) {
9. Properties props = new Properties();
10. props.setProperty("annotators", "tokenize,ssplit, parse");*//pos*
11. *// if you want to use the faster shift reduce parser*
12. *//props.setProperty("parse.model", "edu/stanford/nlp/models/srparser/englishSR.ser.gz");*
13. *//props.setProperty("parse.maxlen", "100");*
14. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
16. Annotation annotation = new Annotation("The small red car turned very quickly around the corner.");
18. pipeline.annotate(annotation);
19. *// get tree from the first sentence*
20. Tree tree = annotation.get(CoreAnnotations.SentencesAnnotation.class).get(0).get(TreeCoreAnnotations.TreeAnnotation.class);
21. System.out.println(tree);
22. */\* Set<Constituent> treeConstituents = tree.constituents(new LabeledScoredConstituentFactory());*
23. *for (Constituent constituent : treeConstituents) {*
24. *//System.out.println(constituent);*
25. *if (constituent.label() != null &&*
26. *(constituent.label().toString().equals("VP") || constituent.label().toString().equals("NP"))) {*
27. *System.out.println("found constituent: "+constituent.toString());*
28. *System.out.println(tree.getLeaves().subList(constituent.start(), constituent.end()+1));*
29. *}*
30. *}\*/*
31. }
32. }

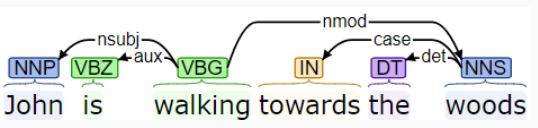
After execution, you should get the below output.

(ROOT (S (NP (DT The) (JJ small) (JJ red) (NN car)) (VP (VBD turned) (ADVP (RB very) (RB quickly)) (PP (IN around) (NP (DT the) (NN corner)))) (. .)))

Try visualizing the tree for the same sentence using Stanford CoreNLP online demo tool.

Dependency Parsing

Dependency Parsing is the technique which analyzes the grammatical structure of a sentence to find the relationships among the words. The relationship is of form 'headword' > 'the relation' > 'modifier'. Here the modifier is dependent on the headword.

For example, in the sentence, 'the black dog is barking at the tree', if we remove the word 'dog', the sentence would not make any sense, but if we remove 'black', the sentence still makes sense. So in the phrase 'black dog', 'black' is dependent on 'dog', or 'black' modifies the word 'dog'. So the headword in this phrase is 'dog' and the dependant or modifier in the phrase is 'black'.

Let us consider the following sentence, 'John is walking towards the woods'. The result of dependency parsing will be as below.

The above structure can also be represented as below:

-> walking/VBG (root)  
    -> John/NNP (nsubj)  
    -> is/VBZ (aux)  
    -> woods/NNS (nmod:towards)  
       -> towards/IN (case)  
       -> the/DT (det)

Let us understand the above dependencies. Typically the main verb in the sentence is the root word or headword for all other relations, so most of the arrows start from the verb. The verb here is 'walking'.  'walking' has three dependents in the above sentence. One is the subject ('John'), the auxiliary ('is') and the modifier ('woods'). In other words, 'John', 'is', and 'woods' modifies the verb 'walking'.

The word 'woods' is present in the phrase 'towards the woods', where 'woods' is the headword and it has 'towards' and 'the' as the dependents. In other words, 'towards' and 'the' modifies the word 'woods'.

Now that you have understood what is dependency parsing, let us understand how to implement this in Stanford CoreNLP.

In Stanford CoreNLP, we can perform dependency parsing using 'depparse' annotator. This annotator provides faster dependency parsing than 'parse' annotator. It is dependent on tokenize, ssplit and pos annotators. Below are the annotations generated by 'parse' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| depparse | DependencyParseAnnotator | BasicDependenciesAnnotation, EnhancedDependenciesAnnotation, EnhancedPlusPlusDependenciesAnnotation |

Some of the properties for the 'depparse' annotator are mentioned below.

| **Property name** | **Type** | **Default value** | **Description** |
| --- | --- | --- | --- |
| depparse.model | file, classpath or URL | edu/stanford/nlp/models/parser/nndep/english\_UD.gz | Using this option, we can set a different model for parsing. |

Stanford NLP provides pre-trained, high-performance neural network dependency parser models for both English and Chinese language. You can download the models from the official website. Moreover, we can also train our own neural network based dependency parser.

Next, let us go through a demo of dependency parsing in Stanford CoreNLP.

Let us look at the below demo. You can try using the Neural Network Dependency Parser.

1. import java.util.Properties;
2. import edu.stanford.nlp.pipeline.CoreDocument;
3. import edu.stanford.nlp.pipeline.CoreSentence;
4. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
5. import edu.stanford.nlp.semgraph.SemanticGraph;
6. public class DepparseDemo {
7. public static void main(String[] args) {
9. Properties props = new Properties();
10. props.setProperty("annotators", "tokenize,ssplit,pos, depparse");
11. String text ="The small red car turned very quickly around the corner.";
12. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
13. CoreDocument document = new CoreDocument(text);
14. pipeline.annotate(document);
15. *// getting the 1st sentence from the annotated document*
16. CoreSentence sentence = document.sentences().get(0);
17. SemanticGraph dependencyParse = sentence.dependencyParse();
18. System.out.println("Example: dependency parse");
19. System.out.println(dependencyParse);
20. }
21. }

After execution, you should get the below output.

Example: dependency parse  
-> turned/VBD (root)  
  -> car/NN (nsubj)  
    -> The/DT (det)  
    -> small/JJ (amod)  
    -> red/JJ (amod)  
  -> quickly/RB (advmod)  
    -> very/RB (advmod)  
  -> corner/NN (nmod:around)  
    -> around/IN (case)  
    -> the/DT (det)  
  -> ./. (punct)

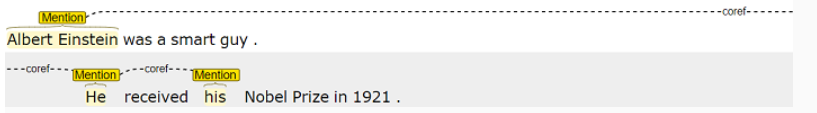
Try visualizing the output for the same sentence using Stanford CoreNLP online demo tool.

Coreference Resolution

In NLP, the process of identifying all the words in a piece of text which refers to the same entity is called coreference resolution. Considering the below sentence;

'Albert Einstein was a smart guy. He received his Nobel Prize in 1921.'

Here the coreference resolution system understands that 'Albert Einstein', 'he' and 'his' refer to one person.



Coreference resolution provides us with the capability to gather all of the information which is available about any particular entity in a piece of text. Coreference resolution is also useful in chatbot applications, where we expect the bot to understand, what or about whom are we talking about in the conversation.

You can make use of the online CoreNLP demo tool to experiment with different texts. Some sentences that you can try are:

* 'John met Jane in 2013. He married her a few years later.' Here 'he' refers to 'John' and 'her' refers to 'Jane'.
* 'Right now I’m testing CoreNLP and its features. It provides an annotator for coreference resolution.' Here 'CoreNLP', 'its' and 'It' refer to the same entity.
* Some sentences might be tricky or ambiguous. For example, 'Right now I’m testing CoreNLP and its features. It‘s always a pleasure to submit tricky texts.' Here, the 'it' in the second sentence refers to the whole sentence. But CoreNLP might give you a wrong reference relating the second 'it' to 'CoreNLP'.

You can experiment with the Stanford NLP Coreference resolution system with some more sentences like below and interpret the results.  
'Luke’s first master was Obi-Wan. He was a good teacher.'  
'The Chicago Bulls could count on Michael Jordan during their six winning seasons, and they are considered one of the strongest teams ever.'

Let us now look at what Stanford CoreNLP offers to perform coreference resolution.

In Stanford CoreNLP, we can perform coreference resolution using 'coref' and 'dcoref' annotators. These annotators are dependent on tokenize, ssplit, pos, lemma, ner, parse (or depparse) annotators.

In CoreNLP, there are three types of coreference resolution systems. They are,

* Deterministic: This is a rule-based, fast coreference resolution system for Chinese and English.
* Statistical: This is a coreference resolution system for English language based on machine learning. This system requires only dependency parsing because it is faster than constituency parsing.
* Neural: This is coreference resolution system is slow but most accurate and it is for Chinese and English language. It is neural network based model.

Neural based coreference resolution is more accurate but it is time consuming. You can find a comparative study of all the three systems in Stanford NLP's official website.

We can choose which algorithm to use by specifying the property 'coref.algorithm' to either 'deterministic' or 'neural' or 'statistical'. Note that we can use 'dcoref' annotator directly to perform rule-based or deterministic coreference resolution. Also, note that the 'statistical' system requires the annotator 'depparse' and the other systems require 'parse'.

Below are the annotations generated by these annotators.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| coref | CorefAnnotator | CorefChainAnnotation |
| dcoref | DeterministicCorefAnnotator | CorefChainAnnotation |

Next, let us see how to implement this in Stanford CoreNLP.

Let us look at the below demo.

1. import java.util.Properties;
2. import edu.stanford.nlp.coref.CorefCoreAnnotations;
3. import edu.stanford.nlp.coref.data.CorefChain;
4. import edu.stanford.nlp.pipeline.Annotation;
5. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
6. public class CorefDemo {
7. public static void main(String[] args) {
9. Annotation document = new Annotation("Mike Tyson is a boxer. He won heavyweight title at 20. Tyson is also a part time actor. Muhammad Ali was an American professional boxer. He is widely regarded as one of the most significant and celebrated sports figures of the 20th century.");
10. Properties props = new Properties();
11. props.setProperty("annotators", "tokenize,ssplit,pos,lemma,ner,parse,coref");
12. props.setProperty("coref.algorithm", "neural");
13. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
14. pipeline.annotate(document);
16. System.out.println("coref chains");
17. for (CorefChain cc : document.get(CorefCoreAnnotations.CorefChainAnnotation.class).values()) {
18. System.out.println(cc);
19. }
20. }
21. }

After execution, you should get the below output.

coref chains  
CHAIN4-["Mike Tyson" in sentence 1, "He" in sentence 2, "Tyson" in sentence 3]  
CHAIN8-["Muhammad Ali" in sentence 4, "He" in sentence 5]

Relation Extraction

Stanford CoreNLP provides us with the utility to extract or identify relationships between entities. We can use the 'relation' annotator for this purpose. But the default model which is used for relation extraction is limited to identifying relations such as 'Live\_In', 'Located\_In', 'OrgBased\_In', 'Work\_For' and 'None'. For identifying other kinds of relations, we can train custom models. To know more on training custom models, refer to the official website.

The 'relation' annotator is dependent on tokenize, ssplit, pos, lemma, ner and parse annotators. Below are the annotations generated by 'relation' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| relation | RelationExtractorAnnotator | MachineReadingAnnotations.RelationMentionsAnnotation |

Once we train our custom model, we can specify the trained model in the Stanford CoreNLP pipeline with property 'sup.relation.model'.

Next, let us understand how to implement this annotator in Stanford CoreNLP.

Let us look at the below demo. Feel free to experiment with the given code.

1. import java.util.List;
2. import java.util.Properties;
3. import edu.stanford.nlp.ie.machinereading.structure.MachineReadingAnnotations;
4. import edu.stanford.nlp.ie.machinereading.structure.RelationMention;
5. import edu.stanford.nlp.ling.CoreAnnotations;
6. import edu.stanford.nlp.pipeline.Annotation;
7. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
8. import edu.stanford.nlp.util.CoreMap;
9. public class RelationDemo {
10. public static void main(String[] args) {
11. Properties props = new Properties();
12. props.setProperty("annotators", "tokenize,ssplit,pos,lemma,ner,parse, relation");
13. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
14. String text = "Barack Obama lives in America. John works for Google. Roger's hometown is Boston.";
16. Annotation doc = new Annotation(text);
17. pipeline.annotate(doc);
19. List<CoreMap> sentences = doc.get(CoreAnnotations.SentencesAnnotation.class);
21. for (CoreMap sentence : sentences) {
22. System.out.println("\n"+sentence);
23. List<RelationMention> relationMentions = sentence.get(MachineReadingAnnotations.RelationMentionsAnnotation.class);
24. *//System.out.println(relationMentions);*
26. for (RelationMention relation : relationMentions) {
27. *//System.out.println(relation);*
28. */\* System.out.println("Start----");*
29. *System.out.println(relation.getArg(0).getType()+" : "+relation.getArg(0).getValue());*
30. *System.out.println(relation.getType()+ " : "+ relation.getTypeProbabilities());*
31. *System.out.println(relation.getArg(1).getType()+" : "+relation.getArg(1).getValue());*
32. *System.out.println("End----");\*/*
34. System.out.println("--------");
35. System.out.println(relation.getArg(0).getValue()+ " -> "+relation.getType()+ " -> "+relation.getArg(1).getValue());
36. }
37. }
38. }
39. }

After execution, you should get the below output.

Barack Obama lives in America.  
--------  
Barack -> Located\_In -> America  
--------  
America -> \_NR -> Barack

John works for Google.  
--------  
John -> OrgBased\_In -> Google  
--------  
Google -> Work\_For -> John

Roger's hometown is Boston.  
--------  
Roger -> Live\_In -> Boston  
--------  
Boston -> Live\_In -> Roger

Note that, the relation annotator gives the relations in both the directions of the entities. Also, you can view the probabilities or confidence values for the five relation categories by uncommenting the given piece of code.

Sentiment Analysis

The process of identifying the emotion associated with a given piece of text is called Sentiment Analysis. This emotion may be neutral, positive or negative. For example, the sentiment associated with the sentence 'I got a good score in the exam' is positive.

Stanford CoreNLP provides the annotator 'sentiment' for the performing sentiment analysis. This annotator is dependent on tokenize, ssplit, pos and parse annotators. It classifies the text into one of the five categories which are 'Very Positive', 'Positive', 'Neutral', 'Negative', 'Very Negative'. It also provides us with scores for each category.

Basic sentiment analysis can be done just by looking at the words used in a text. But if the order of the words is altered, the sentiment associated with the text might change. For example, the words 'funny' and 'witty' are positive. But when you consider the sentence 'This movie was actually neither that funny nor super witty', the sentiment associated is negative. For this purpose, Stanford CoreNLP uses a deep learning based model for sentiment analysis. This model is built on top of the grammatical structures and it is based on Recursive Neural Network. It computes the sentiment based on how words compose the meaning of longer phrases. This is the differentiator from the previous kind of models. With the deep learning based model, the above example will be correctly associated with negative sentiment.

Below are the annotations generated by 'sentiment' annotator.

| **Name** | **Annotator Class Name** | **Generated Annotation** |
| --- | --- | --- |
| sentiment | SentimentAnnotator | SentimentCoreAnnotations.AnnotatedTree |

We can also train our own model on top of the trained model provided by Stanford NLP. We can set the property 'sentiment.model' for using the custom trained model. For more information on training, refer to the official website.

Next, let us see how to perform sentiment analysis on the given text in Stanford CoreNLP.

Let us look at the below demo. Feel free to experiment with the given code.

1. import java.util.Properties;
2. import org.ejml.simple.SimpleMatrix;
3. import edu.stanford.nlp.ling.CoreAnnotations;
4. import edu.stanford.nlp.neural.rnn.RNNCoreAnnotations;
5. import edu.stanford.nlp.pipeline.Annotation;
6. import edu.stanford.nlp.pipeline.StanfordCoreNLP;
7. import edu.stanford.nlp.sentiment.SentimentCoreAnnotations;
8. import edu.stanford.nlp.trees.Tree;
9. import edu.stanford.nlp.util.CoreMap;
10. public class SentimentDemo {
11. public static void main(String[] args) {
13. Annotation document = new Annotation("I am not sure whether to go to the movie or not. After every night, comes a brand new day. I have a headache. The right motivation always helps you to succeed in life.");
15. Properties props = new Properties();
17. props.setProperty("annotators", "tokenize,ssplit,pos,parse, sentiment");
19. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
21. pipeline.annotate(document);
23. for (CoreMap sentence : document.get(CoreAnnotations.SentencesAnnotation.class)) {
24. Tree tree = sentence.get(SentimentCoreAnnotations.SentimentAnnotatedTree.class);
25. SimpleMatrix sm = RNNCoreAnnotations.getPredictions(tree);
26. String sentimentType = sentence.get(SentimentCoreAnnotations.SentimentClass.class);
28. System.out.println("Sentence: "+ sentence);
30. */\*System.out.println("Very Positive: "+(double)Math.round(sm.get(4) \* 100d));*
31. *System.out.println("Positive: "+(double)Math.round(sm.get(3) \* 100d));*
32. *System.out.println("Neutral: "+(double)Math.round(sm.get(2) \* 100d));*
33. *System.out.println("Negative: "+(double)Math.round(sm.get(1) \* 100d));*
34. *System.out.println("Very Negative: "+(double)Math.round(sm.get(0) \* 100d));*
36. *System.out.println("Sentiment Score: "+RNNCoreAnnotations.getPredictedClass(tree));\*/*
37. System.out.println("Sentiment Type: "+sentimentType);
38. System.out.println();
39. }
40. }
41. }

After execution, you should get the below output.

Sentence: I am not sure whether to go to the movie or not.  
Sentiment Type: Negative

Sentence: After every night, comes a brand new day.  
Sentiment Type: Positive

Sentence: I have a headache.  
Sentiment Type: Neutral

Sentence: The right motivation always helps you to succeed in life.  
Sentiment Type: Positive

Some more Annotators

Till now we have learned some important annotators in Stanford CoreNLP. Apart from these, there are many more annotators that we can use. Some of these annotators are mentioned in the table below.

| **Name** | **Annotator Class Name** | **Generated Annotation** | **Description** |
| --- | --- | --- | --- |
| openie | OpenIEAnnotator | EntailedSentencesAnnotation, RelationTriplesAnnotation | It extracts relation triples (subject, relation, object) from the text. The relation here is not predefined, rather it is part of the text. For example, 'Mark Zuckerberg founded Facebook.' Here the output will be Mark Zuckerberg (subject), founded (relation), Facebook (object). |
| kbp | KBPAnnotator | KBPTriplesAnnotation | It also extracts (subject, relation, object) triples from the text, but here the relations are predefined. Some of the relations are org\_founded, org\_website, per\_date\_of\_birth, per\_stateorprovince\_of\_birth etc. |
| quote | QuoteAnnotator | QuotationAnnotation | It extracts quotes from the text. |
| regexner | TokensRegexNERAnnotator | NamedEntityTagAnnotation | It provides a simple, rule-based NER over tokens. It is created using TokensRegex, therefore it is like a simple version of TokensRegex. By default, it recognizes entities such as CITY, NATIONALITY, IDEOLOGY etc. |
| cleanxml | CleanXmlAnnotator | XmlContextAnnotation | This annotator helps in removing xml tokens from the text. It can also be used to extract metadata from text or mark the sentence end. |

Next, let us look at a demo where we use multiple annotators.

In this demo, we are using multiple annotators on the text. You can try experimenting with the code.

1. import edu.stanford.nlp.coref.data.CorefChain;
2. import edu.stanford.nlp.ling.\*;
3. import edu.stanford.nlp.ie.util.\*;
4. import edu.stanford.nlp.pipeline.\*;
5. import edu.stanford.nlp.semgraph.\*;
6. import edu.stanford.nlp.trees.\*;
7. import java.util.\*;
8. public class BasicPipelineExample {
9. public static String text = "Joe Smith was born in California. " +
10. "In 2017, he went to Paris, France in the summer. " +
11. "His flight left at 3:00pm on July 10th, 2017. " +
12. "After eating some escargot for the first time, Joe said, \"That was delicious!\" " +
13. "He sent a postcard to his sister Jane Smith. " +
14. "After hearing about Joe's trip, Jane decided she might go to France one day.";
15. public static void main(String[] args) {
16. *// set up pipeline properties*
17. Properties props = new Properties();
18. *// set the list of annotators to run*
19. props.setProperty("annotators", "tokenize,ssplit,pos,lemma,ner,parse,depparse,coref,kbp,quote");
20. *// setting the property for coref annotator's algorithm to neural network based*
21. props.setProperty("coref.algorithm", "neural");
22. *// create the pipeline object with the specified properties*
23. StanfordCoreNLP pipeline = new StanfordCoreNLP(props);
24. *// create the document object with the given text*
25. CoreDocument document = new CoreDocument(text);
26. *// perform the annotation on the document*
27. pipeline.annotate(document);
28. *// outputs*
29. *// get the tenth token from the annotated document object*
30. CoreLabel token = document.tokens().get(10);
31. System.out.println("Example: token");
32. System.out.println(token);
33. System.out.println();
34. *// get the text from the 1st sentence from the document object*
35. String sentenceText = document.sentences().get(0).text();
36. System.out.println("Example: sentence");
37. System.out.println(sentenceText);
38. System.out.println();
39. *// get the 2nd sentence from the document object*
40. CoreSentence sentence = document.sentences().get(1);
41. *// get the list of pos tags for the 2nd sentence*
42. List<String> posTags = sentence.posTags();
43. System.out.println("Example: pos tags");
44. System.out.println(posTags);
45. System.out.println();
46. *// get the list of ner tags for the 2nd sentence*
47. List<String> nerTags = sentence.nerTags();
48. System.out.println("Example: ner tags");
49. System.out.println(nerTags);
50. System.out.println();
51. *// for the 2nd sentence get the constituency parsing output*
52. Tree constituencyParse = sentence.constituencyParse();
53. System.out.println("Example: constituency parse");
54. System.out.println(constituencyParse);
55. System.out.println();
56. *// for the 2nd sentence get the dependency parsing output*
57. SemanticGraph dependencyParse = sentence.dependencyParse();
58. System.out.println("Example: dependency parse");
59. System.out.println(dependencyParse);
60. System.out.println();
61. *// extract the kbp relations from the 5th sentence*
62. List<RelationTriple> relations = document.sentences().get(4).relations();
63. System.out.println("Example: relation");
64. System.out.println(relations.get(0));
65. System.out.println();
66. *// extract the entity mentions from the 2nd sentence*
67. List<CoreEntityMention> entityMentions = sentence.entityMentions();
68. System.out.println("Example: entity mentions");
69. System.out.println(entityMentions);
70. System.out.println();
71. *// print the coreference between entity mentions found from the 4th sentence*
72. CoreEntityMention originalEntityMention = document.sentences().get(3).entityMentions().get(1);
73. System.out.println("Example: original entity mention");
74. System.out.println(originalEntityMention);
75. System.out.println("Example: canonical entity mention");
76. System.out.println(originalEntityMention.canonicalEntityMention().get());
77. System.out.println();
78. *// getting coref information from the whole document*
79. Map<Integer, CorefChain> corefChains = document.corefChains();
80. System.out.println("Example: coref chains for document");
81. System.out.println(corefChains);
82. System.out.println();
83. *// extract the quotes from the document object*
84. List<CoreQuote> quotes = document.quotes();
85. CoreQuote quote = quotes.get(0);
86. System.out.println("Example: quote");
87. System.out.println(quote);
88. System.out.println();
89. *// get the original speaker details of the quote*
90. System.out.println("Example: original speaker of quote");
91. System.out.println(quote.speaker().get());
92. System.out.println();
93. *// get the canonical speaker details of the quote*
94. System.out.println("Example: canonical speaker of quote");
95. System.out.println(quote.canonicalSpeaker().get());
96. System.out.println();
97. }
98. }

After execution, you should get the below output.

Example: token  
he-4

Example: sentence  
Joe Smith was born in California.

Example: pos tags  
[IN, CD, ,, PRP, VBD, TO, NNP, ,, NNP, IN, DT, NN, .]

Example: ner tags  
[O, DATE, O, O, O, O, CITY, O, COUNTRY, O, O, DATE, O]

Example: constituency parse  
(ROOT (S (PP (IN In) (NP (CD 2017))) (, ,) (NP (PRP he)) (VP (VBD went) (PP (TO to) (NP (NNP Paris) (, ,) (NNP France))) (PP (IN in) (NP (DT the) (NN summer)))) (. .)))

Example: dependency parse  
-> went/VBD (root)  
  -> 2017/CD (nmod:in)  
    -> In/IN (case)  
  -> ,/, (punct)  
  -> he/PRP (nsubj)  
  -> Paris/NNP (nmod:to)  
    -> to/TO (case)  
    -> ,/, (punct)  
    -> France/NNP (appos)  
  -> summer/NN (nmod:in)  
    -> in/IN (case)  
    -> the/DT (det)  
  -> ./. (punct)

Example: relation  
1.0    Jane Smith    per:siblings    Joe Smith

Example: entity mentions  
[2017, Paris, France, summer, he]

Example: original entity mention  
Joe  
Example: canonical entity mention  
Joe Smith

Example: coref chains for document  
{23=CHAIN23-["Joe Smith" in sentence 1, "he" in sentence 2, "His" in sentence 3, "Joe" in sentence 4, "He" in sentence 5, "his" in sentence 5, "Joe 's" in sentence 6], 26=CHAIN26-["his sister Jane Smith" in sentence 5, "Jane" in sentence 6, "she" in sentence 6], 12=CHAIN12-["2017" in sentence 2, "2017" in sentence 3]}

Example: quote  
"That was delicious!"

Example: original speaker of quote  
Joe

Example: canonical speaker of quote  
Joe Smith