Contents

[**To pull up all functions in a module** 2](#_Toc510979746)

[**Natural language processing:** 2](#_Toc510979747)

[**Important terms:** 3](#_Toc510979748)

[**Tokenization** – text into token 3](#_Toc510979749)

[**Tokens** – words or entities present in the text 3](#_Toc510979750)

[**Text** **object** – a sentence or a phrase or a word or an article 3](#_Toc510979751)

[**Text Preprocessing:** 3](#_Toc510979752)

[**Noise removal** (removing stop words, urls, punctuation, mentions etc..) 3](#_Toc510979753)

[**Lexicon normalization** (Tokenization, lemmatization, stemming) 4](#_Toc510979754)

[Stemming:  This process strips the suffixes (“ing”, “ly”, “es”, “s” etc) from a word. 4](#_Toc510979755)

[Lemmatization: Organized & step by step procedure to obtain the root form of the word, 4](#_Toc510979756)

[**Example for lemmatization and stemming** 4](#_Toc510979757)

[**Object standardization** (Regular exp, lookup tables) 5](#_Toc510979758)

[**Encoding-decoding noise:** 5](#_Toc510979759)

[**Grammar checker:** 5](#_Toc510979760)

[**Spelling correction:** 5](#_Toc510979761)

[**Text to Features (Feature Engineering on text data)** 5](#_Toc510979762)

[**Syntactic Parsing** 6](#_Toc510979763)

[Dependency trees 6](#_Toc510979764)

[Part of speech tagging : 7](#_Toc510979765)

[**Word sense disambiguation:** Some language words have multiple meanings according to their usage. 8](#_Toc510979766)

[**Improving word-based features:** 8](#_Toc510979767)

[**Normalization and Lemmatization**: POS tags are the basis of lemmatization process for converting a word to its base form (lemma). 8](#_Toc510979768)

[**Efficient stopword removal** 8](#_Toc510979769)

[Entity Extraction (Entities as features) 8](#_Toc510979770)

[Named Entity Recognition (NER) 9](#_Toc510979771)

[**Noun phrase identification:** 9](#_Toc510979772)

[**Phrase classification:** 9](#_Toc510979773)

[**Entity disambiguation:** 10](#_Toc510979774)

[**Topic Modeling:** Topic modeling is a process of automatically identifying the topics present in a text corpus 10](#_Toc510979775)

[N-Grams as Features: A combination of N words together are called N-Grams. 11](#_Toc510979776)

[Statistical Features 12](#_Toc510979777)

[Term Frequency – Inverse Document Frequency (TF – IDF) 12](#_Toc510979778)

[Count / Density / Readability Features 13](#_Toc510979779)

[Word Embedding (text vectors) 13](#_Toc510979780)

[**Important tasks of NLP** 14](#_Toc510979781)

[**Text Classification(Classify the text object in one fixed category)** 14](#_Toc510979782)

[Text Matching / Similarity 16](#_Toc510979783)

[**Important applications of text matching includes** 17](#_Toc510979784)

[**Levenshtein Distance** 17](#_Toc510979785)

[**Phonetic Matching** 17](#_Toc510979786)

[Flexible String Matching 18](#_Toc510979787)

[Cosine Similarity 18](#_Toc510979788)

[Coreference Resolution 19](#_Toc510979789)

[Other NLP problems / tasks 19](#_Toc510979790)

[Important Libraries for NLP (python) 19](#_Toc510979791)

# **To pull up all functions in a module**

import nltk

for i in dir(nltk): print (i)

# **Natural language processing:**

21% of data is unstructured in the world

Data is being generated as we speak, as we tweet, as we send messages on Whatsapp and in various other activities. Majority of this data exists in the textual form, which is highly unstructured in nature. In order to produce significant and actionable insights from text data, it is important to get acquainted with the techniques and principles of Natural Language Processing (NLP).

**NLP is a branch of data science for systematic processing for**

- analyzing

- Understanding

- deriving information out of text data in smart & efficient manner

**Wide range of problem solving:**

- Automatic summarization

- Machine translation

- named entity recognition

- Relationship extraction

- Sentiment analysis

- Speech recognition

- Topic segmentation etc.

# **Important terms:**

## **Tokenization** – text into token

## **Tokens** – words or entities present in the text

## **Text** **object** – a sentence or a phrase or a word or an article

# **Text Preprocessing:**

This includes three steps for removing noised as it is unstructured

Here we go. Raw data goes through below three process to clean unwanted things.

## **Noise removal** (removing stop words, urls, punctuation, mentions etc..)

|  |
| --- |
| # Sample code to remove noisy words from a text  noise\_list = ["is", "a", "this", "..."]  def \_remove\_noise(input\_text):  words = input\_text.split()  noise\_free\_words = [word for word in words if word not in noise\_list]  noise\_free\_text = " ".join(noise\_free\_words)  return noise\_free\_text  \_remove\_noise("this is a sample text") |

|  |
| --- |
| ```  # Sample code to remove a regex pattern  import re  def \_remove\_regex(input\_text, regex\_pattern):  urls = re.finditer(regex\_pattern, input\_text)  for i in urls:  input\_text = re.sub(i.group().strip(), '', input\_text)  return input\_text  regex\_pattern = "#[\w]\*"  \_remove\_regex("remove this #hashtag from analytics vidhya", regex\_pattern)  >>> "remove this from analytics vidhya"  ``` |

## **Lexicon normalization** (Tokenization, lemmatization, stemming)

Another type of textual noise is about the multiple representations exhibited by single word.

For example – “play”, “player”, “played”, “plays” and “playing” are the different variations of the word – “play”, Though they mean different but contextually all are similar. The step converts all the disparities of a word into their normalized form (also known as lemma).

most common lexicon normalization practices

### Stemming:  This process strips the suffixes (“ing”, “ly”, “es”, “s” etc) from a word.

Lemmatization: Organized & step by step procedure to obtain the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations).

### **Example for lemmatization and stemming**

|  |
| --- |
| ```  from nltk.stem.wordnet import WordNetLemmatizer  lem = WordNetLemmatizer()  from nltk.stem.porter import PorterStemmer  stem = PorterStemmer()  word = "multiplying"  lem.lemmatize(word, "v")  >> "multiply"  stem.stem(word)  >> "multipli"  ``` |

## **Object standardization** (Regular exp, lookup tables)

Text data often contains words or phrases which are not present in any standard lexical dictionaries

Some of the examples are – acronyms, hashtags with attached words, and colloquial slangs. With the help of regular expressions and manually prepared data dictionaries, this type of noise can be fixed, the code below uses a dictionary lookup method to replace social media slangs from a text.

Examle of Object standardization

|  |
| --- |
| ```  lookup\_dict = {'rt':'Retweet', 'dm':'direct message', "awsm" : "awesome", "luv" :"love", "..."}  def \_lookup\_words(input\_text):  words = input\_text.split()  new\_words = []  for word in words:  if word.lower() in lookup\_dict:  word = lookup\_dict[word.lower()]  new\_words.append(word) new\_text = " ".join(new\_words)  return new\_text  \_lookup\_words("RT this is a retweeted tweet by Shivam Bansal")  >> "Retweet this is a retweeted tweet by Shivam Bansal" |

https://www.analyticsvidhya.com/blog/2014/11/text-data-cleaning-steps-python/

## **Encoding-decoding noise:**

## **Grammar checker:**

## **Spelling correction:**

## **Text to Features (Feature Engineering on text data)**

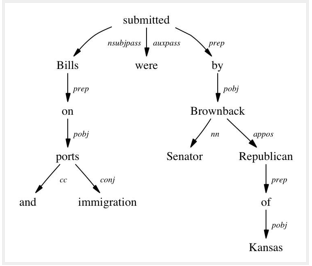
To analyse the preprocessed data. It needs be converted into features.

There are different techniques for this functionality

# **Syntactic Parsing**

## Dependency trees

* Sentences are composed of some words sewed together.
* The relationship among the words in a sentence is determined by the basic dependency grammar.
* Dependency grammar is a class of syntactic text analysis that deals with (labeled) asymmetrical binary relations between two lexical items (words).
* Every relation can be represented in the form of a triplet (relation, governor, dependent). For example: consider the sentence – “Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas.” The relationship among the words can be observed in the form of a tree representation as shown:



* The tree shows that “submitted” is the root word of this sentence, and is linked by two sub-trees (subject and object subtrees).
* Each subtree is a itself a dependency tree with relations such as – (“Bills” <-> “ports” <by> “proposition” relation), (“ports” <-> “immigration” <by> “conjugation” relation).
* This type of tree, when parsed recursively in top-down manner gives grammar relation triplets as output which can be used as features for many nlp problems like entity wise sentiment analysis, actor & entity identification, and text classification.
* The python wrapper [StanfordCoreNLP](http://stanfordnlp.github.io/CoreNLP/) (by Stanford NLP Group, only commercial license) and NLTK dependency grammars can be used to generate dependency trees.

Part of speech tagging :

* Apart from the grammar relations, every word in a sentence is also associated with a part of speech (pos) tag (nouns, verbs, adjectives, adverbs etc) that defines the usage and function of a word in the sentence.
* H ere is a list of all possible pos-tags defined by Pennsylvania university. Following code using NLTK performs pos tagging annotation on input text. (it provides several implementations, the default one is perceptron tagger)

|  |
| --- |
| ```  from nltk import word\_tokenize, pos\_tag  text = "I am learning Natural Language Processing on Analytics Vidhya"  tokens = word\_tokenize(text)  print pos\_tag(tokens)  >>> [('I', 'PRP'), ('am', 'VBP'), ('learning', 'VBG'), ('Natural', 'NNP'),('Language', 'NNP'),  ('Processing', 'NNP'), ('on', 'IN'), ('Analytics', 'NNP'),('Vidhya', 'NNP')]  ``` |

### **Word sense disambiguation:** Some language words have multiple meanings according to their usage.

For example, in the two sentences below:

I. “Please book my flight for Delhi”

II. “I am going to read this book in the flight”

“Book” is used with different context, however the part of speech tag for both of the cases are different. In sentence I, the word “book” is used as **verb**, while in II it is used as **noun**. ([Lesk Algorithm](https://en.wikipedia.org/wiki/Lesk_algorithm) is also used for similar purposes)

### **Improving word-based features:**

A learning model could learn different contexts of a word when used word as the features, however if the part of speech tag is linked with them, the context is preserved, thus making strong features. For example:

**Sentence -“book my flight, I will read this book”**

**Tokens – (“book”, 2), (“my”, 1), (“flight”, 1), (“I”, 1), (“will”, 1), (“read”, 1), (“this”, 1)**

**Tokens with POS – (“book\_VB”, 1), (“my\_PRP$”, 1), (“flight\_NN”, 1), (“I\_PRP”, 1), (“will\_MD”, 1), (“read\_VB”, 1), (“this\_DT”, 1), (“book\_NN”, 1)**

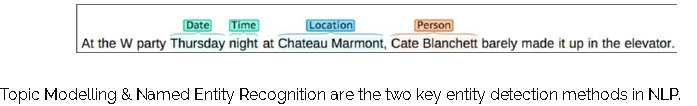
### **Normalization and Lemmatization**: POS tags are the basis of lemmatization process for converting a word to its base form (lemma).

**Efficient stopword removal:** POS tags are also useful in efficient removal of stopwords.

For example, there are some tags which always define the low frequency / less important words of a language. For example: (**IN** – “within”, “upon”, “except”), (**CD** – “one”,”two”, “hundred”), (**MD** – “may”, “must” etc)

## Entity Extraction (Entities as features)

* Entities are defined as the most important chunks of a sentence – noun phrases, verb phrases or both. Entity Detection algorithms are generally ensemble models of rule based parsing, dictionary lookups, pos tagging and dependency parsing.
* **The applicability of entity detection can be seen in the** 
  + automated chat bots
  + content analyzers
  + Consumer insights.



## Named Entity Recognition (NER)

The process of detecting the named entities such as person names, location names, company names etc from the text is called as NER. For example :

**Sentence – Sergey Brin, the manager of Google Inc. is walking in the streets of New York.**

**Named Entities –  ( “person” : “Sergey Brin” ), (“org” : “Google Inc.”), (“location” : “New York”)**

A typical NER model consists of three blocks:

**Noun phrase identification:** This step deals with extracting all the noun phrases from a text using dependency parsing and part of speech tagging.

**Phrase classification:**

* All the extracted noun phrases in the previous step are classified into respective categories (locations, names etc).
* Google Maps API provides a good path to disambiguate locations,
* Then, the open databases from dbpedia, wikipedia can be used to identify person names or company names.
* Apart from this, one can curate the lookup tables and dictionaries by combining information from different sources.

**Entity disambiguation:**

Sometimes it is possible that entities are misclassified, hence creating a validation layer on top of the results is useful. Use of knowledge graphs can be exploited for this purposes. The popular knowledge graphs are –

* Google Knowledge Graph
* IBM Watson
* Wikipedia.

## **Topic Modeling:** Topic modeling is a process of automatically identifying the topics present in a text corpus

* it derives the hidden patterns among the words in the corpus in an unsupervised manner.
* Topics are defined as “a repeating pattern of co-occurring terms in a corpus”.
* A good topic model results in – “health”, “doctor”, “patient”, “hospital” for a topic – Healthcare, and “farm”, “crops”, “wheat” for a topic – “Farming”.

Latent Dirichlet Allocation (LDA) is the most popular topic modelling technique, Following is the code to implement topic modeling using LDA in python. For a detailed explanation about its working and implementation, check the complete article [here.](https://www.analyticsvidhya.com/blog/2016/08/beginners-guide-to-topic-modeling-in-python/)

|  |
| --- |
| ```  doc1 = "Sugar is bad to consume. My sister likes to have sugar, but not my father."  doc2 = "My father spends a lot of time driving my sister around to dance practice."  doc3 = "Doctors suggest that driving may cause increased stress and blood pressure."  doc\_complete = [doc1, doc2, doc3]  doc\_clean = [doc.split() for doc in doc\_complete]  import gensim from gensim  import corpora  # Creating the term dictionary of our corpus, where every unique term is assigned an index.  dictionary = corpora.Dictionary(doc\_clean)  # Converting list of documents (corpus) into Document Term Matrix using dictionary prepared above.  doc\_term\_matrix = [dictionary.doc2bow(doc) for doc in doc\_clean]  # Creating the object for LDA model using gensim library  Lda = gensim.models.ldamodel.LdaModel  # Running and Training LDA model on the document term matrix  ldamodel = Lda(doc\_term\_matrix, num\_topics=3, id2word = dictionary, passes=50)  # Results  print(ldamodel.print\_topics())  ``` |

## N-Grams as Features: A combination of N words together are called N-Grams.

N grams (N > 1) are generally more informative as compared to words (Unigrams) as features. Also, bigrams (N = 2) are considered as the most important features of all the others. The following code generates bigram of a text.

|  |
| --- |
| ```  def generate\_ngrams(text, n):  words = text.split()  output = []  for i in range(len(words)-n+1):  output.append(words[i:i+n])  return output  >>> generate\_ngrams('this is a sample text', 2)  # [['this', 'is'], ['is', 'a'], ['a', 'sample'], , ['sample', 'text']]  ``` |

## Statistical Features

Text data can also be quantified directly into numbers using several techniques described in this section:

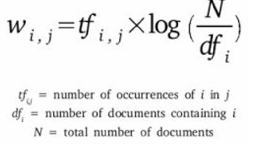
## Term Frequency – Inverse Document Frequency (TF – IDF)

* TF-IDF is a weighted model commonly used for information retrieval problems.
* It aims to convert the text documents into vector models on the basis of occurrence of words in the documents without taking considering the exact ordering.
* For Example – let say there is a dataset of N text documents, In any document “D”, TF and IDF will be defined as –

**Term Frequency (TF)** – TF for a term “t” is defined as the count of a term “t” in a document “D”

**Inverse Document Frequency (IDF**) – IDF for a term is defined as logarithm of ratio of total documents available in the corpus and number of documents containing the term T.

**TF . IDF –** TF IDF formula gives the relative importance of a term in a corpus (list of documents), given by the following formula below. Following is the code using python’s scikit learn package to convert a text into tf idf vectors:



|  |
| --- |
| ```  from sklearn.feature\_extraction.text import TfidfVectorizer  obj = TfidfVectorizer()  corpus = ['This is sample document.', 'another random document.', 'third sample document text']  X = obj.fit\_transform(corpus)  print X  >>>  (0, 1) 0.345205016865  (0, 4) ... 0.444514311537  (2, 1) 0.345205016865  (2, 4) 0.444514311537  ```  The model creates a vocabulary dictionary and assigns an index to each word. Each row in the output contains a tuple (i,j) and a tf-idf value of word at index j in document i. |

## Count / Density / Readability Features

* Count or Density based features can also be used in models and analysis.
* These features might seem trivial but shows a great impact in learning models.
* Some of the features are: Word Count, Sentence Count, Punctuation Counts and Industry specific word counts. Other types of measures include readability measures such as syllable counts, smog index and flesch reading ease.
* Refer to [Textstat](https://github.com/shivam5992/textstat) library to create such features.

## Word Embedding (text vectors)

* Word embedding is the modern way of representing words as vectors and aims to redefine the high dimensional word features into low dimensional feature vectors by preserving the contextual similarity in the corpus.
* They are widely used in deep learning models such as Convolutional Neural Networks and Recurrent Neural Networks.

[Word2Vec](https://code.google.com/archive/p/word2vec/) and [GloVe](http://nlp.stanford.edu/projects/glove/) are the two popular models to create word embedding of a text. These models takes a text corpus as input and produces the word vectors as output.

Word2Vec model is composed of preprocessing module, a shallow neural network model called **Continuous Bag of Words** and another shallow neural network model called **skip-gram**. These models are widely used for all other nlp problems. It first constructs a vocabulary from the training corpus and then learns word embedding representations. Following code using **gensim** package prepares the word embedding as the vectors.

|  |
| --- |
| ```  from gensim.models import Word2Vec  sentences = [['data', 'science'], ['vidhya', 'science', 'data', 'analytics'],['machine', 'learning'], ['deep', 'learning']]  # train the model on your corpus  model = Word2Vec(sentences, min\_count = 1)  print model.similarity('data', 'science')  >>> 0.11222489293  print model['learning']  >>> array([ 0.00459356  0.00303564 -0.00467622  0.00209638, ...])  ``` |

They can be used as feature vectors for ML model, used to measure text similarity using cosine similarity techniques, words clustering and text classification techniques.

## **Important tasks of NLP**

Various use cases in natural language processing

### **Text Classification(Classify the text object in one fixed category)**

**Text classification is one of the classical problem of NLP**

1. Email Spam Identification
2. topic classification of news
3. sentiment classification
4. organization of web pages by search engines.

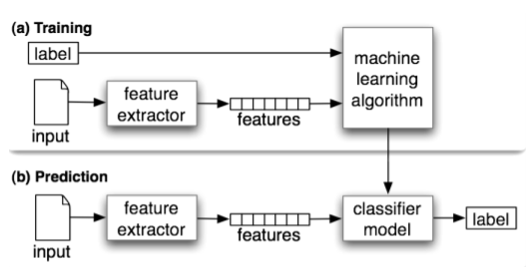
**Text classification, in common words is defined as a technique to systematically classify a text object (document or sentence) in one of the fixed category.** It is really helpful when the amount of data is too large, especially for

* organizing
* information filtering
* storage purposes.

Natural language classifier has two parts

1. Training
2. Prediction

A typical natural language classifier consists of two parts: (a) Training (b) Prediction as shown in image below. Firstly the text input is processes and features are created. The machine learning models then learn these features and is used for predicting against the new text.



|  |
| --- |
| #Here is a code that uses naive bayes classifier using text blob library (built on top of nltk).  ```  from textblob.classifiers import NaiveBayesClassifier as NBC  from textblob import TextBlob  training\_corpus = [  ('I am exhausted of this work.', 'Class\_B'),  ("I can't cooperate with this", 'Class\_B'),  ('He is my badest enemy!', 'Class\_B'),  ('My management is poor.', 'Class\_B'),  ('I love this burger.', 'Class\_A'),  ('This is an brilliant place!', 'Class\_A'),  ('I feel very good about these dates.', 'Class\_A'),  ('This is my best work.', 'Class\_A'),  ("What an awesome view", 'Class\_A'),  ('I do not like this dish', 'Class\_B')]  test\_corpus = [  ("I am not feeling well today.", 'Class\_B'),  ("I feel brilliant!", 'Class\_A'),  ('Gary is a friend of mine.', 'Class\_A'),  ("I can't believe I'm doing this.", 'Class\_B'),  ('The date was good.', 'Class\_A'), ('I do not enjoy my job', 'Class\_B')]  model = NBC(training\_corpus)  print(model.classify("Their codes are amazing."))  >>> "Class\_A"  print(model.classify("I don't like their computer."))  >>> "Class\_B"  print(model.accuracy(test\_corpus))  >>> 0.83  ``` |

Scikit.Learn also provides a pipeline framework for text classification:

|  |
| --- |
| ```  from sklearn.feature\_extraction.text  import TfidfVectorizer from sklearn.metrics  import classification\_report  from sklearn import svm  # preparing data for SVM model (using the same training\_corpus, test\_corpus from naive bayes example)  train\_data = []  train\_labels = []  for row in training\_corpus:  train\_data.append(row[0])  train\_labels.append(row[1])  test\_data = []  test\_labels = []  for row in test\_corpus:  test\_data.append(row[0])  test\_labels.append(row[1])  # Create feature vectors  vectorizer = TfidfVectorizer(min\_df=4, max\_df=0.9)  # Train the feature vectors  train\_vectors = vectorizer.fit\_transform(train\_data)  # Apply model on test data  test\_vectors = vectorizer.transform(test\_data)  # Perform classification with SVM, kernel=linear  model = svm.SVC(kernel='linear')  model.fit(train\_vectors, train\_labels)  prediction = model.predict(test\_vectors)  >>> ['Class\_A' 'Class\_A' 'Class\_B' 'Class\_B' 'Class\_A' 'Class\_A']  print (classification\_report(test\_labels, prediction)) |

The text classification model are heavily dependent upon the quality and quantity of features, while applying any machine learning model it is always a good practice to include more and more training data. H ere are some tips that I wrote about improving the text classification accuracy in one of my previous article.

## Text Matching / Similarity

One of the main areas of NLP to find the matching text object to find similarities.

### **Important applications of text matching includes**

* automatic spelling correction
* data de-duplication
* genome analysis etc.

### **Levenshtein Distance**

**Levenshtein Distance of two strings is defined as the minimum number of edit required to transform one string to another** with the allowable edit operations being **insertion, deletion, or substitution of a single character**

First, install the following:

|  |
| --- |
| pip install editdistance  import editdistance  editdistance.eval(list1, list2) |
| nltk.edit\_distance("aa bbbb cc", "aa b cc") |
| def levenshtein(s1,s2):  if len(s1) > len(s2):  s1,s2 = s2,s1  distances = range(len(s1) + 1)  for index2,char2 in enumerate(s2):  newDistances = [index2+1]  for index1,char1 in enumerate(s1):  if char1 == char2:  newDistances.append(distances[index1])  else:  newDistances.append(1 + min((distances[index1], distances[index1+1], newDistances[-1])))  distances = newDistances  return distances[-1]  print(levenshtein("analyze","analyse")) |

## **Phonetic Matching**

A Phonetic matching algorithm takes a keyword as input (person’s name, location name etc) and produces a character string that identifies a set of words that are (roughly) phonetically similar

Soundex and Metaphone are two main phonetic algorithms used for this purpose. Python’s module Fuzzy is used to compute soundex strings for different words, for example –

|  |
| --- |
| ```  import fuzzy  soundex = fuzzy.Soundex(4)  print soundex('ankit')  >>> “A523”  print soundex('aunkit')  >>> “A523”  ``` |

Flexible String Matching :

* A complete text matching system includes different algorithms pipelined together to compute variety of text variations.
* Regular expressions are really helpful for this purposes as well. Another common techniques include –
  + exact string matching,
  + lemmatized matching,
  + compact matching (takes care of spaces, punctuation’s, slangs etc).

Cosine Similarity

* When the text is represented as vector notation, a general cosine similarity can also be applied in order to measure vectorized similarity.
* Following code converts a text to vectors (using term frequency) and applies cosine similarity to provide closeness among two text.

|  |
| --- |
| import math  from collections import Counter  def get\_cosine(vec1, vec2):  common = set(vec1.keys()) & set(vec2.keys())  numerator = sum([vec1[x] \* vec2[x] for x in common])  sum1 = sum([vec1[x]\*\*2 for x in vec1.keys()])  sum2 = sum([vec2[x]\*\*2 for x in vec2.keys()])  denominator = math.sqrt(sum1) \* math.sqrt(sum2)    if not denominator:  return 0.0  else:  return float(numerator) / denominator  def text\_to\_vector(text):  words = text.split()  return Counter(words)  text1 = 'This is an article on analytics vidhya'  text2 = 'article on analytics vidhya is about natural language processing'  vector1 = text\_to\_vector(text1)  vector2 = text\_to\_vector(text2)  cosine = get\_cosine(vector1, vector2)  >>> 0.62  ``` |

## Coreference Resolution

Coreference Resolution is a process of finding relational links among the words (or phrases) within the sentences. Consider an example sentence: ” Donald went to John’s office to see the new table. He looked at it for an hour.“

Humans can quickly figure out that “he” denotes Donald (and not John), and that “it” denotes the table (and not John’s office). Coreference Resolution is the component of NLP that does this job automatically. It is used in document summarization, question answering, and information extraction. Stanford CoreNLP provides a python [wrapper](https://github.com/Wordseer/stanford-corenlp-python) for commercial purposes.

## Other NLP problems / tasks

* **Text Summarization** – Given a text article or paragraph, summarize it automatically to produce most important and relevant sentences in order.
* **Machine Translation** – Automatically translate text from one human language to another by taking care of grammar, semantics and information about the real world, etc.
* **Natural Language Generation and Understanding** – Convert information from computer databases or semantic intents into readable human language is called language generation. Converting chunks of text into more logical structures that are easier for computer programs to manipulate is called language understanding.
* **Optical Character Recognition** – Given an image representing printed text, determine the corresponding text.
* **Document to Information** – This involves parsing of textual data present in documents (websites, files, pdfs and images) to analyzable and clean format.

## Important Libraries for NLP (python)

* Scikit-learn: Machine learning in Python
* Natural Language Toolkit (NLTK): The complete toolkit for all NLP techniques.
* Pattern – A web mining module for the with tools for NLP and machine learning.
* TextBlob – Easy to use nl p tools API, built on top of NLTK and Pattern.
* spaCy – Industrial strength N LP with Python and Cython.
* Gensim – Topic Modelling for Humans
* Stanford Core NLP – NLP services and packages by Stanford NLP Group.