Ultimate Guide to Understand & Implement Natural Language Processing (with codes in Python)

MACHINE LEARNING PYTHON

SHARE

SHIVAM BANSAL , JANUARY 12, 2017 / 34

According to industry estimates, only 21% of the available data is present in structured form. 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.

Few notorious examples include – tweets / posts on social media, user to user chat conversations, news, blogs and articles, product or services reviews and patient records in the healthcare sector. A few more recent ones includes chatbots and other voice driven bots.

Despite having high dimension data, the information present in it is not directly accessible unless it is processed (read and understood) manually or analyzed by an automated system.

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).

So, if you plan to create chatbots this year, or you want to use the power of unstructured text, this guide is the right starting point. This guide unearths the concepts of natural language processing, its techniques and implementation. The aim of the article is to teach the concepts of natural language processing and apply it on real data set.

Table of Contents

        - Flexible String Matching
    - Coreference Resolution
    - Other Problems
5. Important NLP libraries

1. Introduction to Natural Language Processing

NLP is a branch of data science that consists of systematic processes for analyzing, understanding, and deriving information from the text data in a smart and efficient manner. By utilizing NLP and its components, one can organize the massive chunks of text data, perform numerous automated tasks and solve a wide range of problems such as – automatic summarization, machine translation, named entity recognition, relationship extraction, sentiment analysis, speech recognition, and topic segmentation etc.

Before moving further, I would like to explain some terms that are used in the article:

- Tokenization – process of converting a text into tokens
- Tokens – words or entities present in the text
- Text object – a sentence or a phrase or a word or an article

Steps to install NLTK and its data:

Install Pip: run in terminal:

```
sudo easy_install pip
```

Install NLTK: run in terminal :

```
    sudo pip install -U nltk
```

Download NLTK data: run python shell (in terminal) and write the following code:

```
```


import nltk nltk.download() ```
```

Follow the instructions on screen and download the desired package or collection. Other libraries can be directly installed using pip.

# 2. Text Preprocessing

Since, text is the most unstructured form of all the available data, various types of noise are present in it and the data is not readily analyzable without any pre-processing. The entire process of cleaning and standardization of text, making it noise-free and ready for analysis is known as text preprocessing.

It is predominantly comprised of three steps:

- Noise Removal
- Lexicon Normalization
- Object Standardization

The following image shows the architecture of text preprocessing pipeline.

## 2.1 Noise Removal
```

Any piece of text which is not relevant to the context of the data and the end-output can be specified as the noise.

For example – language stopwords (commonly used words of a language – is, am, the, of, in etc), URLs or links, social media entities (mentions, hashtags), punctuations and industry specific words. This step deals with removal of all types of noisy entities present in the text.

A general approach for noise removal is to prepare a dictionary of noisy entities, and iterate the text object by tokens (or by words), eliminating those tokens which are present in the noise dictionary.

Following is the python code for the same purpose.

```
```

```
# 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 text"

```

Another approach is to use the regular expressions while dealing with special patterns of noise. We have explained regular expressions in detail in one of our previous article. Following python code removes a regex pattern from the input 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"



```

2.2 Lexicon Normalization

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). Normalization is a pivotal step for feature engineering with text as it converts the high dimensional features (N different features) to the low dimensional space (1 feature), which is an ideal ask for any ML model.

The most common lexicon normalization practices are :

- **Stemming:** Stemming is a rudimentary rule-based process of stripping the suffixes (“ing”, “ly”, “es”, “s” etc) from a word.
- **Lemmatization:** Lemmatization, on the other hand, is an organized & step by step procedure of obtaining the root form of the word, it makes use of vocabulary (dictionary importance of words) and morphological analysis (word structure and grammar relations).

Below is the sample code that performs lemmatization and stemming using python’s popular library – NLTK.

```
```

```
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"



```

2.3 Object Standardization

Text data often contains words or phrases which are not present in any standard lexical dictionaries. These pieces are not recognized by search engines and models.

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.

```
```

lookup_dict = {'rt':'Retweet', 'dm':'direct message', "awsm" : "awesome", "luv" :"lov
e", "..."}

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"



```

Apart from three steps discussed so far, other types of text preprocessing includes encoding-decoding noise, grammar checker, and spelling correction etc. The detailed article about preprocessing and its methods is given in one of my previous article**.**

3.Text to Features (Feature Engineering on text data)

To analyse a preprocessed data, it needs to be converted into features. Depending upon the usage, text features can be constructed using assorted techniques – Syntactical

Parsing, Entities / N-grams / word-based features, Statistical features, and word embeddings. Read on to understand these techniques in detail.

3.1 Syntactic Parsing

Syntactical parsing invol ves the analysis of words in the sentence for grammar and their arrangement in a manner that shows the relationships among the words. Dependency Grammar and Part of Speech tags are the important attributes of text syntactics.

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 (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). The pos tags 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')]
```

```
```

Part of Speech tagging is used for many important purposes in NLP:

A.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 ***v erb***, while in II it is used as ***no un***. (Lesk Algorithm is also us ed for similar purposes)

B.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)

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

D.Efficient stopword removal : P OS 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", "mu st" etc)

3.2 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 and consumer insights.

Topic Modelling & Named Entity Recognition are the two key entity detection methods in NLP.

A. 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: This is the classification step in which all the extracted noun phrases 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 and Wikipedia.

B. 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.

```
```

```
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 pr
epared 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())



```

C. 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']]

```

3.3 Statistical Features

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

A. 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 docum
ent 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.

### B. 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 library to create such features.

## 3.4 Word Embedding (text vectors)

Word embedding is the modern way of representing words as vectors. The aim of word embedding is 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 and 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'],['machin
e', '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.

4. Important tasks of NLP

This section talks about different use cases and problems in the field of natural language processing.

4.1 Text Classification

Text classification is one of the classical problem of NLP. Notorious examples include – Email Spam Identification, topic classification of news, sentiment classification and 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, and storage purposes.

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 naiv
e 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.

4.2 Text Matching / Similarity

One of the important areas of NLP is the matching of text objects to find similarities. Important applications of text matching includes automatic spelling correction, data de-duplication and genome analysis etc.

A number of text matching techniques are available depending upon the requirement. This section describes the important techniques in detail.

A. Levenshtein Distance – The Levenshtein distance between two strings is defined as the minimum number of edits needed to transform one string into the other, with the

allowable edit operations being insertion, deletion, or substitution of a single character. Following is the implementation for efficient memory computations.

```
```

```
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"))

```

B. 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. It is very useful for searching large text corpuses, correcting spelling errors and matching relevant names. 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"

```
```

C. 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, and compact matching (takes care of spaces, punctuation's, slangs etc).

D. Cosine Similarity – W hen 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


```

4.3 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 for commercial purposes.

4.4 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.

5. 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.

End Notes

I hope this tutorial will help you maximize your efficiency when starting with natural language processing in Python. I am sure this not only gave you an idea about basic techniques but it also showed you how to implement some of the more sophisticated techniques available today. If you come across any difficulty while practicing Python, or you have any thoughts / suggestions / feedback please feel free to post them in the comments below.

This article was contributed by Shivam Bansal who is the winner of Blogathon 2. We will soon be publishing other top two blogs from the competition Blogathon 2. So, Stay Tuned!