### https://cdn-images-1.medium.com/max/800/1*HgXA9v1EsqlrRDaC_iORhQ.pngA guide to Text Classification (NLP) using SVM and Naive Bayes with Python

I went through a lot of articles, books and videos to understand the text classification technique when I first started it. The content sometimes was too overwhelming for someone who is just beginning with their conquest on NLP or Text Classification Algorithms.

This is my take on explaining the Text classification technique with just the right content to get you started. By the end of this article you will have enough knowledge and a working model to take on the interesting world of Natural Language Processing with Python.

***What is Text Classification?***

Text Classification is an automated process of classification of text into predefined categories. We can classify Emails into spam or non-spam, news articles into different categories like Politics, Stock Market, Sports, etc.

This can be done with the help of Natural Language Processing and different Classification Algorithms like Naive Bayes, SVM and even Neural Networks in Python.

**What is Natural Language Processing?**

Short for ***n****atural* ***l****anguage* ***p****rocessing*, NLP is a branch of artificial intelligence which is focused on the enabling the computers to understand and interpret the human language. The problem with interpreting the human language is that it is not a set of rules or binary data that can be fed into the system and understanding the context of a conversation or reading between the lines is altogether a different ball game.

However, with the recent advancement in Machine Learning, Deep Learning with the help of Neural Networks and easy to use models in python has opened the doors for us to code our way into making computers understand the complex human Language.

**Now let’s realize this with a supervised ML model to classify text:**

I will be using the Amazon Review Data set which has 10,000 rows of Text data which is classified into “Label 1” and “Label 2”. The Data set has two columns “Text” and “Label”. You can download the data from [here](https://github.com/Gunjitbedi/Text-Classification/blob/master/corpus.csv).

***STEP -1****:* ***Add the Required Libraries***

The following libraries will be used ahead in the article. If not available, these can be easily downloaded through their respective websites.

import pandas as pd  
import numpy as np  
from nltk.tokenize import word\_tokenize  
from nltk import pos\_tag  
from nltk.corpus import stopwords  
from nltk.stem import WordNetLemmatizer  
from sklearn.preprocessing import LabelEncoder  
from collections import defaultdict  
from nltk.corpus import wordnet as wn  
from sklearn.feature\_extraction.text import TfidfVectorizer  
from sklearn import model\_selection, naive\_bayes, svm  
from sklearn.metrics import accuracy\_score

***STEP -2****:* ***Set random seed***

This is used to reproduce the same result every time if the script is kept consistent otherwise each run will produce different results. The seed can be set to any number.

np.random.seed(500)

**STEP -3: Add the Corpus**

The data set can be easily added as a pandas Data Frame with the help of ‘read\_csv’ function. I have set the encoding to ‘latin-1’ as the text had many special characters.

Corpus = pd.read\_csv(r"C:\Users\gunjit.bedi\Desktop\NLP Project\corpus.csv",encoding='latin-1')

**STEP -4: Data pre-processing**

This is an important step in any data mining process. This basically involves transforming raw data into an understandable format for NLP models. Real-world data is often incomplete, inconsistent, and/or lacking in certain behaviors or trends, and is likely to contain many errors. Data pre-processing is a proven method of resolving such issues. This will help in getting better results through the classification algorithms.

Below, I have explained the two techniques that are also performed besides other easy to understand steps in data pre-processing:

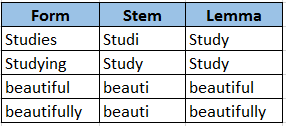
1. **Tokenization**: This is a process of breaking a stream of text up into words, phrases, symbols, or other meaningful elements called tokens. The list of tokens becomes input for further processing. NLTK Library has *word\_tokenize* and *sent\_tokenize* to easily break a stream of text into a list of words or sentences, respectively.
2. **Word Stemming/Lemmatization**: The aim of both processes is the same, reducing the inflectional forms of each word into a common base or root. Lemmatization is closely related to stemming. The difference is that a stemmer operates on a single word without knowledge of the context, and therefore cannot discriminate between words which have different meanings depending on part of speech. However, stemmers are typically easier to implement and run faster, and the reduced accuracy may not matter for some applications.

Image: Lemma performs better

Here’s the complete script which performs the aforementioned data pre-processing steps, you can always add or remove steps which best suits the data set you are dealing with:

1. Remove Blank rows in Data, if any
2. Change all the text to lower case
3. Word Tokenization
4. Remove Stop words
5. Remove Non-alpha text
6. Word Lemmatization

# Step - a : Remove blank rows if any.  
Corpus['text'].dropna(inplace=True)

# Step - b : Change all the text to lower case. This is required as python interprets 'dog' and 'DOG' differently  
Corpus['text'] = [entry.lower() for entry in Corpus['text']]

# Step - c : Tokenization : In this each entry in the corpus will be broken into set of words  
Corpus['text']= [word\_tokenize(entry) for entry in Corpus['text']]

# Step - d : Remove Stop words, Non-Numeric and perfom Word Stemming/Lemmenting.

# WordNetLemmatizer requires Pos tags to understand if the word is noun or verb or adjective etc. By default it is set to Noun  
tag\_map = defaultdict(lambda : wn.NOUN)  
tag\_map['J'] = wn.ADJ  
tag\_map['V'] = wn.VERB  
tag\_map['R'] = wn.ADV

for index,entry in enumerate(Corpus['text']):  
 # Declaring Empty List to store the words that follow the rules for this step  
 Final\_words = []  
 # Initializing WordNetLemmatizer()  
 word\_Lemmatized = WordNetLemmatizer()  
 # pos\_tag function below will provide the 'tag' i.e if the word is Noun(N) or Verb(V) or something else.  
 for word, tag in pos\_tag(entry):  
 # Below condition is to check for Stop words and consider only alphabets  
 if word not in stopwords.words('english') and word.isalpha():  
 word\_Final = word\_Lemmatized.lemmatize(word,tag\_map[tag[0]])  
 Final\_words.append(word\_Final)  
 # The final processed set of words for each iteration will be stored in 'text\_final'  
 Corpus.loc[index,'text\_final'] = str(Final\_words)

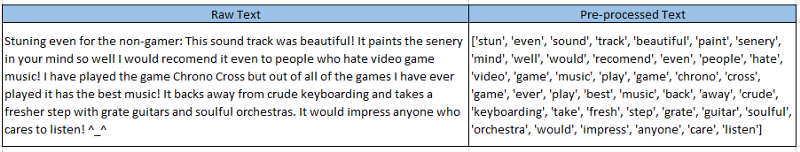


Image: Text after all the pre-processing steps are performed

**STEP -5: Prepare Train and Test Data sets**

The Corpus will be split into two data sets, Training and Test. The training data set will be used to fit the model and the predictions will be performed on the test data set. This can be done through the *train\_test\_split* from the *sklearn* library. The Training Data will have 70% of the corpus and Test data will have the remaining 30% as we have set the parameter *test\_size=0.3 .*

Train\_X, Test\_X, Train\_Y, Test\_Y = model\_selection.train\_test\_split(Corpus['text\_final'],Corpus['label'],test\_size=0.3)

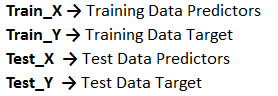


Image: Content of each Data Set

**STEP -6: Encoding**

Label encode the target variable — This is done to transform Categorical data of string type in the data set into numerical values which the model can understand.

Encoder = LabelEncoder()  
Train\_Y = Encoder.fit\_transform(Train\_Y)  
Test\_Y = Encoder.fit\_transform(Test\_Y)

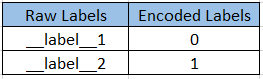


Image: Text Encoding

**STEP -7: Word Vectorization**

It is a general process of turning a collection of text documents into numerical feature vectors. There are many methods to convert text data to vectors which the model can understand but by far the most popular method is called [*TF-IDF*](https://en.wikipedia.org/wiki/Tf%E2%80%93idf). This is an acronym than stands for “*Term Frequency — Inverse Document*” Frequency which are the components of the resulting scores assigned to each word.

* **Term Frequency**: This summarizes how often a given word appears within a document.
* **Inverse Document Frequency**: This down scales words that appear a lot across documents.

Without going into the math, TF-IDF are word frequency scores that try to highlight words that are more interesting, e.g. frequent in a document but not across documents.

The following syntax can be used to first fit the TG-IDF model on the whole corpus. This will help TF-IDF build a vocabulary of words which it has learned from the corpus data and it will assign a unique integer number to each of these words. There will be maximum of 5000 unique words/features as we have set parameter *max\_features=5000.*

Finally, we will transform *Train\_X* and *Test\_X* to vectorized *Train\_X\_Tfidf* and *Test\_X\_Tfidf*. These will now contain for each row a list of unique integer number and its associated importance as calculated by TF-IDF.

Tfidf\_vect = TfidfVectorizer(max\_features=5000)  
Tfidf\_vect.fit(Corpus['text\_final'])

Train\_X\_Tfidf = Tfidf\_vect.transform(Train\_X)  
Test\_X\_Tfidf = Tfidf\_vect.transform(Test\_X)

You can use the below syntax to see the vocabulary that it has learned from the corpus

print(Tfidf\_vect.vocabulary)

This will give an output as

{‘even’: 1459, ‘sound’: 4067, ‘track’: 4494, ‘beautiful’: 346, ‘paint’: 3045, ‘mind’: 2740, ‘well’: 4864, ‘would’: 4952, ‘recommend’: 3493, ‘people’: 3115, ‘hate’: 1961, ‘video’: 4761 …………}

And you can directly print the vectorized data to see how it looks like

print(Train\_X\_Tfidf)



Image: — 1: Row number of ‘Train\_X\_Tfidf’, 2: Unique Integer number of each word in the first row, 3: Score calculated by TF-IDF Vectorizer

Now our data sets are ready to be fed into different classification Algorithms.

**STEP -7: Use the ML Algorithms to Predict the outcome**

First up, let’s try the Naive Bayes Classifier Algorithm. You can read more about it [here](https://en.wikipedia.org/wiki/Naive_Bayes_classifier)

# fit the training dataset on the NB classifier  
Naive = naive\_bayes.MultinomialNB()  
Naive.fit(Train\_X\_Tfidf,Train\_Y)

# predict the labels on validation dataset  
predictions\_NB = Naive.predict(Test\_X\_Tfidf)

# Use accuracy\_score function to get the accuracy  
print("Naive Bayes Accuracy Score -> ",accuracy\_score(predictions\_NB, Test\_Y)\*100)

Output:

Naive Bayes Accuracy Score -> 83.1%

Next is the SVM — Support Vector Machine. You can read more about it [here](https://en.wikipedia.org/wiki/Support_vector_machine)

# Classifier - Algorithm - SVM  
# fit the training dataset on the classifier  
SVM = svm.SVC(C=1.0, kernel='linear', degree=3, gamma='auto')  
SVM.fit(Train\_X\_Tfidf,Train\_Y)

# predict the labels on validation dataset  
predictions\_SVM = SVM.predict(Test\_X\_Tfidf)

# Use accuracy\_score function to get the accuracy  
print("SVM Accuracy Score -> ",accuracy\_score(predictions\_SVM, Test\_Y)\*100)

Output:

SVM Accuracy Score -> 84.6%

I hope this has explained well what text classification is and how it can be easily implemented in Python. If you want the full code you can access it from [here](https://github.com/Gunjitbedi/Text-Classification).

As a next step you can try the following:

1. Play around with the Data pre-processing steps and see how it effects the accuracy.
2. Try other Word Vectorization techniques such as Count Vectorizer and Word2Vec.
3. Try Parameter tuning with the help of GridSearchCV on these Algorithms.
4. Try other classification Algorithms like Linear Classifier, Boosting Models and even Neural Networks.