07-classification-for-text-analysis

October 21, 2021

1 Classification for Text Analysis

[1]:
```
import nltk
```

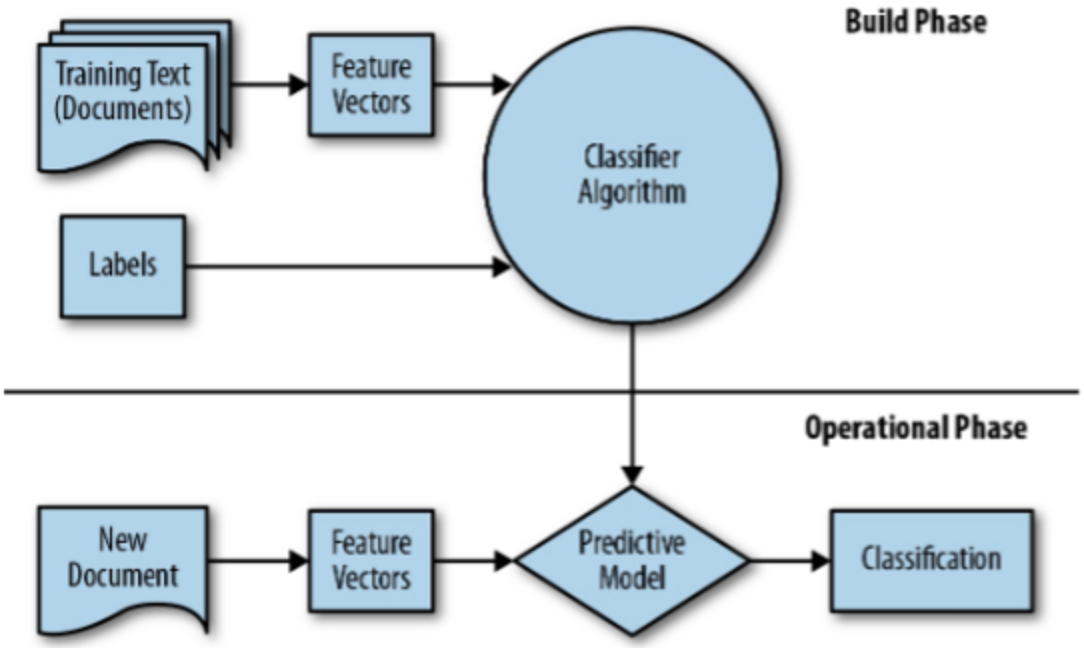
Classification is a primary form of text analysis and is widely used in a variety of domains and applications. The premise of classification is simple: given a categorical target variable, learn patterns that exist between instances composed of independent variables and their relationship to the target. Because the target is given ahead of time, classification is said to be supervised machine learning because a model can be trained to minimize error between predicted and actual categories in the training data. Once a classification model is fit, it assigns categorical labels to new instances based on the patterns detected during training.

This simple premise gives the opportunity for a huge number of possible applications, so long as the application problem can be formulated to identify either a yes/no (binary classification) or discrete buckets (multiclass classification). The most difficult part of applied text analytics is the curation and collection of a domain-specific corpus to build models upon.

All classifier model families have the same basic workflow, and with Scikit-Learn Estimator objects, they can be employed in a procedural fashion and compared using cross-validation to select the best performing predictor.

The classification workflow occurs in two phases: a build phase and an operational phase.

- In the build phase, a corpus of documents is transformed into feature vectors. The document features, along with their annotated labels (the category or class we want the model to learn), are then passed into a classification algorithm that defines its internal state along with the learned patterns.
- Once trained or fitted, a new document can be vectorized into the same space as the training data and passed to the predictive algorithm, which returns the assigned class label for the document.



This is a very practical topic so let's start studying some concrete examples right away. In the rest of this section, we will look at how classifiers can be employed to solve a wide variety of tasks. Our discussion is not intended to be comprehensive, but to give a representative sample of tasks that can be performed with the help of text classifiers.

In any case remember that choosing an appropriate classification algorithm for a particular problem task requires practice and experience; each algorithm has its own quirks and is based on certain assumptions. To restate the **no free lunch theorem** by David H. Wolpert, ***no single classifier works best across all possible scenarios*** *(The Lack of A Priori Distinctions Between Learning Algorithms, Wolpert, David H, Neural Computation 8.7 (1996): 1341- 1390)*. In practice, it is always recommended that you **compare the performance of at least a handful of different learning algorithms to select the best model for the particular problem**.

1.1 Example 1 - Gender Identification

Reference: *Bird S. et al. "Natural Language Processing with Python" O'Reilly (2009), Chapter 6 - "Learning to Classify Text"* and references therein. Here the on-line version.

Focus

- what is a feature in text analysis
- training Vs validation

In this first example we will try to build a simple algorithm to understand if a noun passed in input is of masculine or feminine gender. In this, and in the following examples, we will implicitly assume that the language used is **English**.

Usually one of the first step in creating a classifier is deciding what features of the input are relevant, and how to encode those features. For this example, we'll start by just **looking at the final**

letter of a given name. The following **feature extractor** function builds a dictionary containing relevant information about a given name:

[2]:
```
def gender_features_1(word):
    return {'last_letter': word[-1]}
```

The returned **dictionary**, known as a **feature set**, maps from feature names to their values. Feature names are case-sensitive strings that typically provide a short human-readable description of the feature, as in the example **'last_letter'**. Feature values are values with simple types, such as booleans, numbers, and strings, in this case the last letter of the name.

[3]:
```
gender_features_1("batman")
```

[3]:
```
{'last_letter': 'n'}
```

Now that we've defined a feature extractor, we need to prepare a list of examples and corresponding class labels. For this we are going to use the corpus 'names' which is part of nltk corpus collection. In particular the names corpus contains a total of around 2943 male (male.txt) and 5001 female (female.txt) names.

[4]:
```
from nltk.corpus import names

print("\nNumber of male names:")
print (len(names.words('male.txt')))
print("\nNumber of female names:")
print (len(names.words('female.txt')))

male_names   = names.words('male.txt')
female_names = names.words('female.txt')

print("\nFirst 10 male names:")
print (male_names[0:15])
print("\nFirst 10 female names:")
print (female_names[0:15])
```

```
Number of male names:
2943

Number of female names:
5001

First 10 male names:
['Aamir', 'Aaron', 'Abbey', 'Abbie', 'Abbot', 'Abbott', 'Abby', 'Abdel',
'Abdul', 'Abdulkarim', 'Abdullah', 'Abe', 'Abel', 'Abelard', 'Abner']

First 10 female names:
['Abagael', 'Abagail', 'Abbe', 'Abbey', 'Abbi', 'Abbie', 'Abby', 'Abigael',
```

```
'Abigail', 'Abigale', 'Abra', 'Acacia', 'Ada', 'Adah', 'Adaline']
```

Associate to each name the corresponding label ('male' or 'female') and shuffle the set disrupting alphabetical order

[5]:
```
import random

labeled_names = ([(name, 'male') for name in male_names] +
                 [(name, 'female') for name in female_names])
random.shuffle(labeled_names)
```

[6]:
```
labeled_names[:15]
```

[6]:
```
[('Giovanna', 'female'),
 ('Micheal', 'male'),
 ('Allie', 'female'),
 ('Russel', 'male'),
 ('Vere', 'female'),
 ('Kirsti', 'female'),
 ('Cathleen', 'female'),
 ('Flemming', 'male'),
 ('Teressa', 'female'),
 ('Reed', 'male'),
 ('Stillmann', 'male'),
 ('Bernadina', 'female'),
 ('Jody', 'female'),
 ('Carlotta', 'female'),
 ('Arly', 'female')]
```

Next, we use the feature extractor to process the names data

[7]:
```
featuresets = [(gender_features_1(n), gender) for (n, gender) in labeled_names]
```

[8]:
```
featuresets[:10]
```

[8]:
```
[({'last_letter': 'a'}, 'female'),
 ({'last_letter': 'l'}, 'male'),
 ({'last_letter': 'e'}, 'female'),
 ({'last_letter': 'l'}, 'male'),
 ({'last_letter': 'e'}, 'female'),
 ({'last_letter': 'i'}, 'female'),
 ({'last_letter': 'n'}, 'female'),
 ({'last_letter': 'g'}, 'male'),
 ({'last_letter': 'a'}, 'female'),
 ({'last_letter': 'd'}, 'male')]
```

and divide the resulting list of feature sets into a training set and a test set

[9]:
```
train_set, test_set = featuresets[500:], featuresets[:500]
```

The training set is used to train a "naive Bayes" classifier.

[10]:
```
classifier = nltk.NaiveBayesClassifier.train(train_set)
```

[11]:
```
classifier.classify(gender_features_1('Frodo'))
```

[11]:
```
'male'
```

[12]:
```
classifier.classify(gender_features_1('Trinity'))
```

[12]:
```
'female'
```

We can systematically evaluate the classifier on a much larger quantity of unseen data:

[13]:
```
print(nltk.classify.accuracy(classifier, test_set))
```

```
0.76
```

Selecting relevant features and deciding how to encode them for a learning method can have an enormous impact on the learning method's ability to extract a good model. Much of the interesting work in building a classifier is deciding what features might be relevant, and how we can represent them. Although it's often possible to get decent performance by using a fairly simple and obvious set of features, there are usually significant gains to be had by using carefully constructed features based on a thorough understanding of the task at hand.

Typically, feature extractors are built through a process of trial-and-error, guided by intuitions about what information is relevant to the problem. For example, it is clear that some suffixes that are more than one letter can be indicative of name genders. Names ending in 'yn' appear to be predominantly female (in english), despite the fact that names ending in 'n' tend to be male; and names ending in 'ch' are usually male, even though names that end in 'h' tend to be female.

We therefore adjust our feature extractor to include features for two-letter suffixes:

[14]:
```
def gender_features_2(word):
    return {'suffix1': word[-1:],
            'suffix2': word[-2:]}
```

In this case we are going to use three set. First, we select a development set, containing the corpus data for creating the model. This development set is then subdivided into the training set and the dev-test set. The training set is used to train the model, and the dev-test set is used to perform error analysis. The test set serves in our final evaluation of the system.

[15]:
```
train_names   = labeled_names[1500:]
devtest_names = labeled_names[500:1500]

train_set     = [(gender_features_2(n), gender) for (n, gender) in train_names]
devtest_set   = [(gender_features_2(n), gender) for (n, gender) in devtest_names]
```

```
test_names    = labeled_names[:500]
```

[16]:
```
classifier  = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, devtest_set))
```

```
0.771
```

And what about the 'test_names' dataset? Well, this analysis procedure can then be repeated, checking for patterns in the errors that are made by the newly improved classifier. Each time the error analysis procedure is repeated, we should select a different dev-test/training split, to ensure that the classifier does not start to reflect idiosyncrasies in the dev-test set.

But once we've used the dev-test set to help us develop the model, we can no longer trust that it will give us an accurate idea of how well the model would perform on new data. It is therefore important to keep the test set separate, and unused, until our model development is complete. At that point, we can use the test set to evaluate how well our model will perform on new input values.

1.2 Example 2 - The «20 Newsgroup» dataset

References:

- *Javed Shaikh, "Machine Learning, NLP: Text Classification using scikit-learn, python and NLTK"* Towards Data Science and references therein.
- Scikit-Learn web site

Focus

- using sklearn package
- numerical features and vectorization process
- use of a Naive Bayes classifier
- building a ml pipeline with sklearn
- model optimization

In this example we are going to work with the "20 Newsgroup Dataset". The 20 Newsgroups data set is a collection of approximately 20,000 newsgroup documents, partitioned (nearly) evenly across 20 different newsgroups. The 20 newsgroups collection has become a popular data set for experiments in text applications of machine learning techniques, such as text classification and text clustering. Here you can find the home page of the project and here the description of the scikit inclusion. This data set is infact in-built in scikit, so we don't need to download it explicitly.

1.2.1 Loading the data set

First of all load the training sample:

[17]:
```
from sklearn.datasets import fetch_20newsgroups
twenty_train = fetch_20newsgroups(subset='train', shuffle=True)
```

[18]:
```
twenty_train.target_names #prints all the categories
```

[18]:
```
['alt.atheism',
 'comp.graphics',
 'comp.os.ms-windows.misc',
 'comp.sys.ibm.pc.hardware',
 'comp.sys.mac.hardware',
 'comp.windows.x',
 'misc.forsale',
 'rec.autos',
 'rec.motorcycles',
 'rec.sport.baseball',
 'rec.sport.hockey',
 'sci.crypt',
 'sci.electronics',
 'sci.med',
 'sci.space',
 'soc.religion.christian',
 'talk.politics.guns',
 'talk.politics.mideast',
 'talk.politics.misc',
 'talk.religion.misc']
```

[19]:
```
print("\n".join(twenty_train.data[0].split("\n")[:3])) #prints first line of the␣
 ↪first data file
```

```
From: lerxst@wam.umd.edu (where's my thing)
Subject: WHAT car is this!?
Nntp-Posting-Host: rac3.wam.umd.edu
```

1.2.2 Extracting features from text files

We already known that in order to run machine learning algorithms we need to convert the text files into numerical feature vectors. We will be using bag of words model for our example. Briefly, we segment each text file into words (for English splitting by space), and count # of times each word occurs in each document and finally assign each word an integer id. **Each unique word in our dictionary will correspond to a feature (descriptive feature)**.

Remember that, scikit-learn has a high level component which will create feature vectors for us 'CountVectorizer'. More about it here. Let's see a very simple example of 'CountVectorizer' before apply it to our dataset:

[20]:
```
corpus = [
     'This is the first document.',
     'This document is the second document.',
     'And this is the third one.',
     'Is this the first document?',
]
```

[21]:
```
from sklearn.feature_extraction.text import CountVectorizer
```

[22]:
```
vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)
```

Here by doing 'vectorizer.fit_transform(corpus)', we are learning the vocabulary dictionary and it returns a Document-Term matrix of dimension [n_samples, n_features].

[23]:
```
print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

[24]:
```
print(X.toarray())
```

```
[[0 1 1 1 0 0 1 0 1]
 [0 2 0 1 0 1 1 0 1]
 [1 0 0 1 1 0 1 1 1]
 [0 1 1 1 0 0 1 0 1]]
```

Now apply this vectorizer to our dataset...

[25]:
```
count_vect = CountVectorizer()

X_train_counts = count_vect.fit_transform(twenty_train.data)
X_train_counts.shape
```

[25]:
```
(11314, 130107)
```

We can use also a measure based on Term Frequency matrix. In particular remember two important metrics of this type for Bag-of-Words model:

- **TF**: Just counting the number of words in each document has 1 issue: it will give more weightage to longer documents than shorter documents. To avoid this, we can use frequency (TF - Term Frequencies) i.e. #count(word) / #Total words, in each document.
- **TF-IDF**: Finally, we can even reduce the weightage of more common words like (the, is, an etc.) which occurs in all document. This is called as TF-IDF i.e Term Frequency times inverse document frequency.

Remember that the goal of using tf-idf instead of the raw frequencies of occurrence of a token in a given document is to scale down the impact of tokens that occur very frequently in a given corpus and that are hence empirically less informative than features that occur in a small fraction of the training corpus.

We can perform the calculation of TF-IDF matrix using below line of code:

[26]:
```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names())
```

```
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
```

[27]:
```
print(X.toarray())
```

```
[[0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]
 [0.         0.6876236  0.         0.28108867 0.         0.53864762
  0.28108867 0.         0.28108867]
 [0.51184851 0.         0.         0.26710379 0.51184851 0.
  0.26710379 0.51184851 0.26710379]
 [0.         0.46979139 0.58028582 0.38408524 0.         0.
  0.38408524 0.         0.38408524]]
```

Below we use the 'TfidfTransformer' which transform a count matrix to a normalized tf or tf-idf representation

[28]:
```
from sklearn.feature_extraction.text import TfidfTransformer

tfidf_transformer = TfidfTransformer()

X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
X_train_tfidf.shape
```

[28]:
```
(11314, 130107)
```

1.2.3 Running ML algorithms.

There are various algorithms which can be used for text classification. We will start with the most simplest one *Naive Bayes (NB)*

Naive Bayes

[29]:
```
from sklearn.naive_bayes import MultinomialNB

features    = X_train_tfidf
labels      = twenty_train.target
classifier  = MultinomialNB().fit(features, labels)
```

This will train the NB classifier on the training data we provided.

[30]:
```
twenty_test = fetch_20newsgroups(subset='test', shuffle=True)

X_test_counts = count_vect.transform(twenty_test.data)         # transform metod␣
↪NOT fit_transform
X_test_tfidf  = tfidf_transformer.transform(X_test_counts)
X_test_tfidf.shape
```

[30]:
```
(7532, 130107)
```

Important : Note that for the test set we use the method 'transform' and not 'fit_transform' as before. In layman's terms the reason is that fit_transform means to do some calculation and then do transformation (say calculating the means of columns from some data and then replacing the missing values). So for training set, you need to both calculate and do transformation. But for testing set, the model applies prediction based on what was learned during the training set and so it doesn't need to calculate, it just performs the transformation. You can avoid many of these problems building a 'pipeline' as described above.

[31]:
```
predicted = classifier.predict(X_test_tfidf)
```

Now we will test the performance of the NB classifier on test set:

[32]:
```
import numpy as np
np.mean(predicted == twenty_test.target)
```

[32]: 0.7738980350504514

Building a Pipeline The machine learning process often combines a series of transformers on raw data, transforming the dataset each step of the way until it is passed to the fit method of a final estimator. Pipeline objects enable us to integrate a series of transformers that combine normalization, vectorization, and feature analysis into a single, well-defined mechanism.

The purpose of a Pipeline is to chain together multiple estimators representing a fixed sequence of steps into a single unit. All estimators in the pipeline, except the last one, must be transformers — that is, implement the transform method, while the last estimator can be of any type, including predictive estimators. Pipelines provide convenience; fit and transform can be called for single inputs across multiple objects at once. Pipelines also provide a single interface for grid search of multiple estimators at once. Most importantly, pipelines provide operationalization of text models by coupling a vectorization methodology with a predictive model. Pipelines are constructed by describing a list of (key, value) pairs where the key is a string that names the step and the value is the estimator object. Pipelines can be created either by using the make_pipeline helper function, which automatically determines the names of the steps, or by specifying them directly. Generally, it is better to specify the steps directly to provide good user documentation, whereas make_pipeline is used more often for automatic pipeline construction.

To create a pipeline with sklearn we simply define the steps in the object I have called text_clf in the code below. We can then simply call fit on this object to train the model. The pipeline object can also be used to make predictions on new data.

[33]:
```
from sklearn.pipeline import Pipeline

text_clf = Pipeline([('vect', CountVectorizer()),
                     ('tfidf', TfidfTransformer()),
                     ('clf', MultinomialNB()),])
```

[34]:
```
text_clf = text_clf.fit(twenty_train.data, twenty_train.target)
```

[35]:
```
twenty_test = fetch_20newsgroups(subset='test', shuffle=True)
predicted   = text_clf.predict(twenty_test.data)

np.mean(predicted == twenty_test.target)
```

[35]:
```
0.7738980350504514
```

Removing stop words: (the, then etc) from the data. You should do this only when stop words are not useful for the underlying problem. In most of the text classification problems, this is indeed not useful. Let's see if removing stop words increases the accuracy. Update the code for creating object of CountVectorizer as follows:

[36]:
```
text_clf = Pipeline([('vect', CountVectorizer(stop_words='english')),
                     ('tfidf', TfidfTransformer()),
                     ('clf', MultinomialNB()),])

text_clf = text_clf.fit(twenty_train.data, twenty_train.target)

twenty_test = fetch_20newsgroups(subset='test', shuffle=True)
predicted   = text_clf.predict(twenty_test.data)

np.mean(predicted == twenty_test.target)
```

[36]:
```
0.8169144981412639
```

This is a case in which we have a great improvement removing stopwords from our corpus.

1.2.4 Model optimisation

All estimators in the Scikit-learn library contain a range of parameters, for which there are multiple options. The values that you choose for a particular algorithm will impact how well the final model performs. For example, with the RandomForestClassifier you can set the max_depth of the tree to potentially any value, and depending on your data and task, different values for this parameter will produce different results.

This process of trying different combinations of parameters to find the optimal combination is known as hyperparameter optimisation. Scikit-learn provides two tools to automatically perform this task, **GridSearchCV** which implements a technique known as exhaustive grid search and *RandomizedSearchCV* which performs randomized parameter optimisation.

The below example uses GridSearchCV to find the optimal parameters of our simple Naive Bayes model.

[37]:
```
from sklearn.model_selection import GridSearchCV

parameters = {'vect__ngram_range': [(1, 1), (1, 2)],
              'tfidf__use_idf': (True, False),
              'clf__alpha': (1e-2, 1e-3),}
```

Here, we are creating a list of parameters for which we would like to do performance tuning. All the parameters name start with the classifier name (remember the arbitrary name we gave). E.g. vect__ngram_range; here we are telling to use unigram and bigrams and choose the one which is optimal. Next, we create an instance of the grid search by passing the classifier, parameters and n_jobs=-1 which tells to use multiple cores from user machine.

[38]:
```
gs_clf = GridSearchCV(text_clf, parameters, n_jobs=-1)
gs_clf = gs_clf.fit(twenty_train.data, twenty_train.target)
```

This might take few minutes to run depending on the machine configuration. Lastly, to see the best mean score and the params, run the following code:

[39]:
```
gs_clf.best_score_
```

[39]:
```
0.9129399133330441
```

[40]:
```
gs_clf.best_params_
```

[40]:
```
{'clf__alpha': 0.01, 'tfidf__use_idf': True, 'vect__ngram_range': (1, 2)}
```

The accuracy has now increased to ~90.6% for the NB classifier and the corresponding parameters are {'clf__alpha': 0.01, 'tfidf__use_idf': True, 'vect__ngram_range': (1, 2)}. alpha is the so-called Laplace Smoothing, for more infos see here.

1.3 Example 4 - Amazon Review Data set

Reference: *Gunjit Bedi, "A guide to Text Classification(NLP) using SVM and Naive Bayes with Python", Medium.com and references therein.*

[41]:
```
from nltk.tokenize                    import word_tokenize
from nltk                             import pos_tag
from nltk.corpus                      import stopwords
from nltk.corpus                      import wordnet as wn
from nltk.stem                        import WordNetLemmatizer

from collections                      import defaultdict

from sklearn.preprocessing            import LabelEncoder
from sklearn.feature_extraction.text  import TfidfVectorizer
from sklearn                          import model_selection, naive_bayes, svm
from sklearn.metrics                  import accuracy_score
```

We will use the an 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.

[42]:
```
Corpus = pd.read_csv("./corpus/amazon_corpus.csv",encoding='latin-1')
Corpus.head()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-42-389ed32716bd> in <module>
----> 1 Corpus = pd.read_csv("./corpus/amazon_corpus.csv",encoding='latin-1')
      2 Corpus.head()

NameError: name 'pd' is not defined
```

1.3.1 Data pre-processing

[ ]:
```
# 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
# This a simple mapping from the output of pos_tag function and the tag format␣
 ↪expected by the lemmatize method
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)
```

1.3.2 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)
```

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)
```

1.3.3 Word Vectorization

[ ]:
```
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)
```

[ ]:
```
#print(Tfidf_vect.vocabulary_)
```

[ ]:
```
#print(Train_X_Tfidf)
```

1.3.4 Use the ML Algorithms to Predict the outcome

[ ]:
```
# 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)
```

[ ]:
```
# 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)
```

1.4 Example 5 - Sentiment Analysis Deep Learning with Keras

See notebooks 07-DLP-classifying-movie-revies both from the book of ***François Chollet "Deep Learning with Python"***:

click to open

1.5 References and Credits

Bengfort B. et al., *"Applied Text Analysis with Python"* O'Reilly (2018)

Bird S. et al., *"Natural Language Processing with Python"* O'Reilly (2009)

Chollet F., *"Deep Learning with Python"* Manning (2018)