Published in Analytics Vidhya · Follow

Vijaya Rani · Follow
Mar 31, 2021 · 9 min read

NLP Tutorial for Text Classification in Python

Unstructured data in the form of text: chats, emails, social media, survey responses is present everywhere today. Text can be a rich source of information, but due to its unstructured nature it can be hard to extract insights from it.

Text classification is one of the important task in supervised machine learning (ML). It is a process of assigning tags/categories to documents helping us to automatically & quickly structure and analyze text in a cost-effective manner. It is one of the fundamental tasks in **Natural Language Processing** with broad applications such as sentiment-analysis, spam-detection, topic-labeling, intent-detection etc.

In this article, I would like to take you through the step by step process of how we can do text classification using Python. I have uploaded the complete code on GitHub: https://github.com/vijayaiitk/NLP-text-classification-model

Let's divide the classification problem into the below steps:

1. Setup: Importing Libraries
2. Loading the data set & Exploratory Data Analysis
3. Text pre-processing
4. Extracting vectors from text (Vectorization)
5. Running ML algorithms
6. Conclusion

```
#for text pre-processing
import re, string
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import SnowballStemmer
from nltk.corpus import wordnet
from nltk.stem import WordNetLemmatizer

nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')

#for model-building
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import classification_report, f1_score, accuracy_score, confusion_matrix
from sklearn.metrics import roc_curve, auc, roc_auc_score

# bag of words
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.feature_extraction.text import CountVectorizer

#for word embedding
import gensim
from gensim.models import Word2Vec
```

Step 2: Loading the data set & EDA

The data set that we will be using for this article is the famous "***Natural Language Processing with Disaster Tweets***" data set where we'll be predicting whether a given tweet is about a real disaster (target=1) or not (target=0)

In this competition, you're challenged to build a machine learning model that predicts which Tweets are about real disasters and which ones aren't. You'll have access to a dataset of 10,000 tweets that were hand classified.

Loading the data set in Kaggle Notebook:

```
df_train= pd.read_csv('../input/nlp-getting-started/train.csv')
df_test=pd.read_csv('../input/nlp-getting-started/test.csv')
```

We have 7,613 tweets in training (labelled) dataset and 3,263 in the test(unlabelled) dataset. Here's a snapshot of the training/labelled dataset which we'll use for building our model

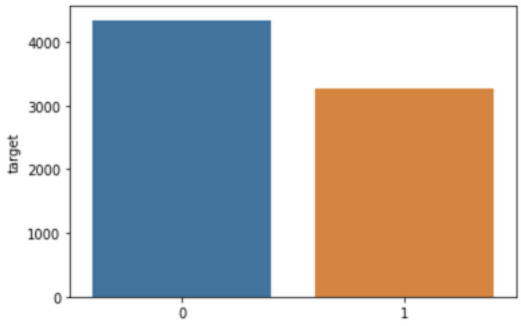
	id	keyword	location	text	target
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1

Snapshot of labelled dataset

Exploratory Data Analysis (EDA)

applying data-balancing techniques like SMOTE while building the model

```
x=df_train['target'].value_counts()
print(x)
sns.barplot(x.index,x)
```



Class distribution

2. ***Missing values***: We have ~2.5k missing values in location field and 61 missing values in keyword column

```
df_train.isna().sum()
```

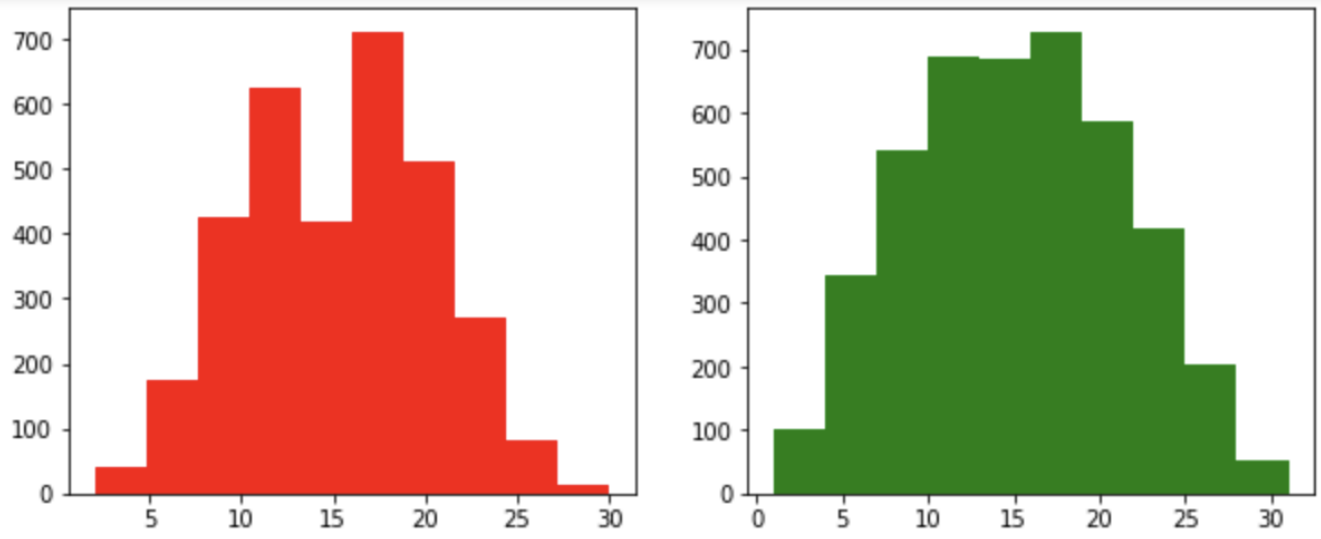
```
id             0
keyword       61
location    2533
text           0
target         0
dtype: int64
```

3. ***Number of words in a tweet***: Disaster tweets are more wordy than the non-disaster tweets

```
# WORD-COUNT
df_train['word_count'] = df_train['text'].apply(lambda x: len(str(x).split()))
print(df_train[df_train['target']==1]['word_count'].mean()) #Disaster tweets
print(df_train[df_train['target']==0]['word_count'].mean()) #Non-Disaster tweets
```

The average number of words in a disaster tweet is 15.17 as compared to an average of 14.7 words in a non-disaster tweet

```
# PLOTTING WORD-COUNT
fig,(ax1,ax2)=plt.subplots(1,2,figsize=(10,4))
train_words=df_train[df_train['target']==1]['word_count']
ax1.hist(train_words,color='red')
ax1.set_title('Disaster tweets')
train_words=df_train[df_train['target']==0]['word_count']
ax2.hist(train_words,color='green')
ax2.set_title('Non-disaster tweets')
fig.suptitle('Words per tweet')
plt.show()
```



Plot for Words per tweet (disaster vs non-disaster tweets)

4. ***Number of characters in a tweet***: Disaster tweets are longer than the non-disaster tweets

```
# CHARACTER-COUNT
df_train['char_count'] = df_train['text'].apply(lambda x: len(str(x)))
print(df_train[df_train['target']==1]['char_count'].mean()) #Disaster tweets
print(df_train[df_train['target']==0]['char_count'].mean()) #Non-Disaster tweets
```

The average characters in a disaster tweet is 108.1 as compared to an average of 95.7 characters in a non-disaster tweet

Step 3: Text Pre-Processing

Before we move to model building, we need to preprocess our dataset by removing punctuations & special characters, cleaning texts, removing stop words, and applying lemmatization

Simple text cleaning processes: Some of the common text cleaning process involves:

- Removing punctuations, special characters, URLs & hashtags
- Removing leading, trailing & extra white spaces/tabs
- Typos, slangs are corrected, abbreviations are written in their long forms

Stop-word removal: We can remove a list of generic stop words from the English vocabulary using *nltk*. A few such words are 'i','you','a','the','he','which' etc.

Stemming: Refers to the process of slicing the end or the beginning of words with the intention of removing affixes(prefix/suffix)

Lemmatization: It is the process of reducing the word to its base form

word	stemming	lemmatization
information	inform	information
having	hav	have
am	am	be
computers	comput	computer

Stemming vs Lemmatization

```
#convert to lowercase, strip and remove punctuations
def preprocess(text):
    text = text.lower()
    text=text.strip()
    text=re.compile('<.*?>').sub('', text)
    text = re.compile('[%s]' % re.escape(string.punctuation)).sub(' ', text)
    text = re.sub('\s+', ' ', text)
    text = re.sub(r'\[[0-9]*\]',' ',text)
    text=re.sub(r'[^\w\s]', '', str(text).lower().strip())
    text = re.sub(r'\d',' ',text)
    text = re.sub(r'\s+',' ',text)
    return text



# STOPWORD REMOVAL
def stopword(string):
    a= [i for i in string.split() if i not in stopwords.words('english')]
    return ' '.join(a)

#LEMMATIZATION
# Initialize the lemmatizer
wl = WordNetLemmatizer()

# This is a helper function to map NTLK position tags
def get_wordnet_pos(tag):
    if tag.startswith('J'):
        return wordnet.ADJ
    elif tag.startswith('V'):
        return wordnet.VERB
    elif tag.startswith('N'):
        return wordnet.NOUN
    elif tag.startswith('R'):
        return wordnet.ADV
    else:
        return wordnet.NOUN

# Tokenize the sentence
def lemmatizer(string):
    word_pos_tags = nltk.pos_tag(word_tokenize(string)) # Get position tags
    a=[wl.lemmatize(tag[0], get_wordnet_pos(tag[1])) for idx, tag in enumerate(word_pos_tags)] # Map the
position tag and lemmatize the word/token
    return " ".join(a)
```

Final pre-processing

```
def finalpreprocess(string):
    return lemmatizer(stopword(preprocess(string)))

df_train['clean_text'] = df_train['text'].apply(lambda x: finalpreprocess(x))
df_train.head()
```

	id	keyword	location	text	target	clean_text
0	1	NaN	NaN	Our Deeds are the Reason of this #earthquake M...	1	deed reason earthquake may allah forgive u
1	4	NaN	NaN	Forest fire near La Ronge Sask. Canada	1	forest fire near la ronge sask canada
2	5	NaN	NaN	All residents asked to 'shelter in place' are ...	1	resident ask shelter place notify officer evac...
3	6	NaN	NaN	13,000 people receive #wildfires evacuation or...	1	people receive wildfire evacuation order calif...
4	7	NaN	NaN	Just got sent this photo from Ruby #Alaska as ...	1	get sent photo ruby alaska smoke wildfires pou...


process to convert text data into numerical data/vector, is called **vectorization** or in the NLP world, word embedding**.** ***Bag-of-Words****(BoW)* and ***Word Embedding*** *(*with Word2Vec) are two well-known methods for converting text data to numerical data.

There are a few versions of **Bag of Words**, corresponding to different words scoring methods. We use the Sklearn library to calculate the BoW numerical values using these approaches:

1. **Count vectors**: It builds a vocabulary from a corpus of documents and counts how many times the words appear in each document

	love	programming	also
1 → I love programming	1	1	0
2 → Programming also loves me	1	1	1

Count vectors

2. **Term Frequency-Inverse Document Frequencies (tf-Idf)**: Count vectors might not be the best representation for converting text data to numerical data. So, instead of simple counting, we can also use an advanced variant of the Bag-of-Words that uses the **term frequency–inverse document frequency** (or Tf-Idf)**.** Basically, the value of a word increases proportionally to count in the document, but it is inversely proportional to the frequency of the word in the corpus

Word2Vec: One of the major drawbacks of using **Bag-of-words** techniques is that it can't capture the meaning or relation of the words from vectors. Word2Vec is one of the most popular technique to learn word embeddings using shallow neural network which is capable of capturing context of a word in a document, semantic and syntactic similarity, relation with other words, etc.

We can use any of these approaches to convert our text data to numerical form which will be used to build the classification model. With this in mind, I am going to first partition the dataset into training set (80%) and test set (20%) using the below-mentioned code

```
#SPLITTING THE TRAINING DATASET INTO TRAIN AND TEST
X_train, X_test, y_train, y_test =
train_test_split(df_train["clean_text"],df_train["target"],test_size=0.2,shuffle=True)

#Word2Vec
# Word2Vec runs on tokenized sentences
X_train_tok= [nltk.word_tokenize(i) for i in X_train]
X_test_tok= [nltk.word_tokenize(i) for i in X_test]
```

Here's the code for vectorization using ***Bag-of-Words*** (with Tf-Idf) and **Word2Vec**

```
#Tf-Idf
tfidf_vectorizer = TfidfVectorizer(use_idf=True)
X_train_vectors_tfidf = tfidf_vectorizer.fit_transform(X_train)
X_test_vectors_tfidf = tfidf_vectorizer.transform(X_test)

#building Word2Vec model
class MeanEmbeddingVectorizer(object):
    def __init__(self, word2vec):
        self.word2vec = word2vec
        # if a text is empty we should return a vector of zeros
        # with the same dimensionality as all the other vectors
        self.dim = len(next(iter(word2vec.values())))

def fit(self, X, y):
        return self

def transform(self, X):
        return np.array([
```

```
w2v = dict(zip(model.wv.index2word, model.wv.syn0)) df['clean_text_tok']=[nltk.word_tokenize(i) for i in
df['clean_text']]
model = Word2Vec(df['clean_text_tok'],min_count=1)
modelw = MeanEmbeddingVectorizer(w2v)

# converting text to numerical data using Word2Vec
X_train_vectors_w2v = modelw.transform(X_train_tok)
X_val_vectors_w2v = modelw.transform(X_test_tok)
```

Step 5. Running ML algorithms

It's time to train a **machine learning model** on the vectorized dataset and test it. Now that we have converted the text data to numerical data, we can run ML models on ***X_train_vector_tfidf*** & ***y_train***. We'll test this model on ***X_test_vectors_tfidf*** to get ***y_predict*** and further evaluate the performance of the model

1. **Logistic Regression:** We will start with the most simplest one Logistic Regression. You can easily build a LogisticRegression in scikit using below lines of code

```
#FITTING THE CLASSIFICATION MODEL using Logistic Regression(tf-idf)

lr_tfidf=LogisticRegression(solver = 'liblinear', C=10, penalty = 'l2')
lr_tfidf.fit(X_train_vectors_tfidf, y_train)

#Predict y value for test dataset
y_predict = lr_tfidf.predict(X_test_vectors_tfidf)
y_prob = lr_tfidf.predict_proba(X_test_vectors_tfidf)[:,1]

print(classification_report(y_test,y_predict))
print('Confusion Matrix:',confusion_matrix(y_test, y_predict))

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
print('AUC:', roc_auc)
```

```
              precision    recall  f1-score   support

           0       0.82      0.84      0.83       857
           1       0.79      0.76      0.77       666

    accuracy                           0.81      1523
   macro avg       0.80      0.80      0.80      1523
weighted avg       0.81      0.81      0.81      1523

Confusion Matrix: [[723 134]
 [160 506]]
AUC: 0.8646537435919
```

```
#FITTING THE CLASSIFICATION MODEL using Logistic Regression (W2v)
lr_w2v=LogisticRegression(solver = 'liblinear', C=10, penalty = 'l2')
lr_w2v.fit(X_train_vectors_w2v, y_train)  #model

#Predict y value for test dataset
y_predict = lr_w2v.predict(X_test_vectors_w2v)
y_prob = lr_w2v.predict_proba(X_test_vectors_w2v)[:,1]

print(classification_report(y_test,y_predict))
print('Confusion Matrix:',confusion_matrix(y_test, y_predict))

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
print('AUC:', roc_auc)
```

```
              accuracy                           0.63      1523
             macro avg       0.62      0.59      0.59      1523
          weighted avg       0.62      0.63      0.60      1523

Confusion Matrix: [[712 159]
 [412 240]]
AUC: 0.6552953730638924
```

2. **Naive Bayes:** It's a probabilistic classifier that makes use of Bayes' Theorem, a rule that uses probability to make predictions based on prior knowledge of conditions that might be related

```
#FITTING THE CLASSIFICATION MODEL using Naive Bayes(tf-idf)

nb_tfidf = MultinomialNB()
nb_tfidf.fit(X_train_vectors_tfidf, y_train)

#Predict y value for test dataset
y_predict = nb_tfidf.predict(X_test_vectors_tfidf)
y_prob = nb_tfidf.predict_proba(X_test_vectors_tfidf)[:,1]

print(classification_report(y_test,y_predict))
print('Confusion Matrix:',confusion_matrix(y_test, y_predict))

fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr)
print('AUC:', roc_auc)
```

```
              precision    recall  f1-score   support

           0       0.78      0.90      0.83       871
           1       0.82      0.66      0.73       652

    accuracy                           0.79      1523
   macro avg       0.80      0.78      0.78      1523
weighted avg       0.80      0.79      0.79      1523

Confusion Matrix: [[780  91]
 [224 428]]
AUC: 0.8445135694815211
```

You can now select the best model (*lr_tfidf* in our case) to estimate 'target' values for the unlabelled dataset (*df_test*). Here's the code for that

```
#Pre-processing the new dataset
df_test['clean_text'] = df_test['text'].apply(lambda x: finalpreprocess(x)) #preprocess the data
X_test=df_test['clean_text']

#converting words to numerical data using tf-idf
X_vector=tfidf_vectorizer.transform(X_test)

#use the best model to predict 'target' value for the new dataset
y_predict = lr_tfidf.predict(X_vector)
y_prob = lr_tfidf.predict_proba(X_vector)[:,1]
df_test['predict_prob']= y_prob
df_test['target']= y_predict
final=df_test[['clean_text','target']].reset_index(drop=True)
print(final.head())
```

1	heard earthquake different city stay safe ever...	1
2	forest fire spot pond geese flee across street...	1
3	apocalypse light spokane wildfire	1
4	typhoon soudelor kill china taiwan	1
5	shake earthquake	1
6	probably still show life arsenal yesterday eh eh	0

Predicted target value for unlabelled dataset

Conclusion

In this article, I demonstrated the basics of building a text classification model comparing **Bag-of-Words** (with Tf-Idf) and **Word Embedding** with Word2Vec. You can further enhance the performance of your model using this code by

- using other classification algorithms like **Support Vector Machines (SVM), XgBoost, Ensemble models, Neural networks etc.**
- using Gridsearch to tune the hyperparameters of your model
- using advanced word-embedding methods like **GloVe** and **BERT**

Kindly like, comment and share if you liked this article. Your feedback is welcome!

vijayaiitk/NLP-text-classification-model
Contribute to vijayaiitk/NLP-text-classification-model development by creating an account on GitHub.
github.com

Vijaya Rani - Business Intelligence Engineer - Amazon | LinkedIn
Master's Candidate in Business Analytics and Information Management with 4-years of experience in Business Strategy &...
www.linkedin.com