StackAbuse

Text Classification with Python and Scikit-Learn

Usman Malik

Introduction

Text classification is one of the most important tasks in Natural Language Processing. It is the process of classifying text strings or documents into different categories, depending upon the contents of the strings. Text classification has a variety of applications, such as detecting user sentiment from a tweet, classifying an email as spam or ham, classifying blog posts into different categories, automatic tagging of customer queries, and so on.

In this article, we will see a real-world example of text classification. We will train a machine learning model capable of predicting whether a given movie review is positive or negative. This is a classic example of sentimental analysis where people's sentiments towards a particular entity are classified into different categories.

Dataset

The dataset that we are going to use for this article can be downloaded from the Cornell Natural Language Processing Group. The dataset consists of a total of 2000 documents. Half of the documents contain positive reviews regarding a movie while the remaining half contains negative reviews. Further details regarding the dataset can be found at this link.

Unzip or extract the dataset once you download it. Open the folder "txt_sentoken". The folder contains two subfolders: "neg" and "pos". If you open these folders, you can see the text documents containing movie reviews.

Sentiment Analysis with Scikit-Learn

Now that we have downloaded the data, it is time to see some action. In this section, we will perform a series of steps required to predict sentiments from reviews of different movies. These steps can be used for any text classification task. We will use Python's Scikit-Learn library for machine learning to train a text classification model.

Following are the steps required to create a text classification model in Python:

1. Importing Libraries
2. Importing The dataset
3. Text Preprocessing
4. Converting Text to Numbers
5. Training and Test Sets
6. Training Text Classification Model and Predicting Sentiment
7. Evaluating The Model
8. Saving and Loading the Model

Importing Libraries

Execute the following script to import the required libraries:

```
import numpy as np
import re
import nltk
from sklearn.datasets import load_files
nltk.download('stopwords')
import pickle
from nltk.corpus import stopwords
```

Importing the Dataset

We will use the `load_files` function from the `sklearn_datasets` library to import the dataset into our application. The `load_files` function automatically divides the dataset

into data and target sets. For instance, in our case, we will pass it the path to the "txt_sentoken" directory. The `load_files` will treat each folder inside the "txt_sentoken" folder as one category and all the documents inside that folder will be assigned its corresponding category.

Execute the following script to see `load_files` function in action:

```
movie_data = load_files(r"D:\txt_sentoken")
X, y = movie_data.data, movie_data.target
```

In the script above, the `load_files` function loads the data from both "neg" and "pos" folders into the `X` variable, while the target categories are stored in `y`. Here `X` is a list of 2000 string type elements where each element corresponds to single user review. Similarly, `y` is a numpy array of size 2000. If you print `y` on the screen, you will see an array of 1s and 0s. This is because, for each category, the `load_files` function adds a number to the target numpy array. We have two categories: "neg" and "pos", therefore 1s and 0s have been added to the target array.

Text Preprocessing

Once the dataset has been imported, the next step is to preprocess the text. Text may contain numbers, special characters, and unwanted spaces. Depending upon the problem we face, we may or may not need to remove these special characters and numbers from text. However, for the sake of explanation, we will remove all the special characters, numbers, and unwanted spaces from our text. Execute the following script to preprocess the data:

```
documents = []

from nltk.stem import WordNetLemmatizer

stemmer = WordNetLemmatizer()

for sen in range(0, len(X)):
    # Remove all the special characters
    document = re.sub(r'\W', ' ', str(X[sen]))

    # remove all single characters
    document = re.sub(r'\s+[a-zA-Z]\s+', ' ', document)

    # Remove single characters from the start
    document = re.sub(r'\^[a-zA-Z]\s+', ' ', document)

    # Substituting multiple spaces with single space
    document = re.sub(r'\s+', ' ', document, flags=re.I)

    # Removing prefixed 'b'
    document = re.sub(r'^b\s+', '', document)

    # Converting to Lowercase
    document = document.lower()

    # Lemmatization
    document = document.split()

    document = [stemmer.lemmatize(word) for word in document]
    document = ' '.join(document)

    documents.append(document)
```

In the script above we use Regex Expressions from Python re library to perform different preprocessing tasks. We start by removing all non-word characters such as special characters, numbers, etc.

Next, we remove all the single characters. For instance, when we remove the punctuation mark from "David's" and replace it with a space, we get "David" and a single character "s", which has no meaning. To remove such single characters we use `\s+[a-zA-Z]\s+`

regular expression which substitutes all the single characters having spaces on either side, with a single space.

Next, we use the `\^[a-zA-Z]\s+` regular expression to replace a single character from the beginning of the document, with a single space. Replacing single characters with a single space may result in multiple spaces, which is not ideal.

We again use the regular expression `\s+` to replace one or more spaces with a single space. When you have a dataset in bytes format, the alphabet letter "b" is appended before every string. The regex `^b\s+` removes "b" from the start of a string. The next step is to convert the data to lower case so that the words that are actually the same but have different cases can be treated equally.

The final preprocessing step is the lemmatization. In lemmatization, we reduce the word into dictionary root form. For instance "cats" is converted into "cat". Lemmatization is done in order to avoid creating features that are semantically similar but syntactically different. For instance, we don't want two different features named "cats" and "cat", which are semantically similar, therefore we perform lemmatization.

Converting Text to Numbers

Machines, unlike humans, cannot understand the raw text. Machines can only see numbers. Particularly, statistical techniques such as machine learning can only deal with numbers. Therefore, we need to convert our text into numbers.

Different approaches exist to convert text into the corresponding numerical form. The Bag of Words Model and the Word Embedding Model are two of the most commonly used approaches. In this article, we will use the bag of words model to convert our text to numbers.

Bag of Words

The following script uses the bag of words model to convert text documents into corresponding numerical features:

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(max_features=1500, min_df=5, max_df=0.7, stop_w
X = vectorizer.fit_transform(documents).toarray()
```

The script above uses `CountVectorizer` class from the `sklearn.feature_extraction.text` library. There are some important parameters that are required to be passed to the constructor of the class. The first parameter is the `max_features` parameter, which is set to 1500. This is because when you convert words to numbers using the bag of words approach, all the unique words in all the documents are converted into features. All the documents can contain tens of thousands of unique words. But the words that have a very low frequency of occurrence are unusually not a good parameter for classifying documents. Therefore we set the `max_features` parameter to 1500, which means that we want to use 1500 most occurring words as features for training our classifier.

The next parameter is `min_df` and it has been set to 5. This corresponds to the minimum number of documents that should contain this feature. So we only include those words that occur in at least 5 documents. Similarly, for the `max_df` , feature the value is set to 0.7; in which the fraction corresponds to a percentage. Here 0.7 means that we should include only those words that occur in a maximum of 70% of all the documents. Words that occur in almost every document are usually not suitable for classification because they do not provide any unique information about the document.

Finally, we remove the stop words from our text since, in the case of sentiment analysis, stop words may not contain any useful information. To remove the stop words we pass the `stopwords` object from the `nltk.corpus` library to the `stop_words` parameter.

The `fit_transform` function of the `CountVectorizer` class converts text documents into corresponding numeric features.

Finding TFIDF

The bag of words approach works fine for converting text to numbers. However, it has one drawback. It assigns a score to a word based on its occurrence in a particular document. It doesn't take into account the fact that the word might also be having a high frequency of occurrence in other documents as well. TFIDF resolves this issue by multiplying the term frequency of a word by the inverse document frequency. The TF stands for "Term Frequency" while IDF stands for "Inverse Document Frequency".

The term frequency is calculated as:

```
Term frequency = (Number of Occurrences of a word)/(Total words in the docum
```

And the Inverse Document Frequency is calculated as:

```
IDF(word) = Log((Total number of documents)/(Number of documents containing
```

The TFIDF value for a word in a particular document is higher if the frequency of occurrence of that word is higher in that specific document but lower in all the other documents.

To convert values obtained using the bag of words model into TFIDF values, execute the following script:

```
from sklearn.feature_extraction.text import TfidfTransformer
tfidfconverter = TfidfTransformer()
X = tfidfconverter.fit_transform(X).toarray()
```

Note:

You can also directly convert text documents into TFIDF feature values (without first converting documents to bag of words features) using the following script:

```
from sklearn.feature_extraction.text import TfidfVectorizer
tfidfconverter = TfidfVectorizer(max_features=1500, min_df=5, max_df=0.7, st
X = tfidfconverter.fit_transform(documents).toarray()
```

Training and Testing Sets

Like any other supervised machine learning problem, we need to divide our data into training and testing sets. To do so, we will use the `train_test_split` utility from the `sklearn.model_selection` library. Execute the following script:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, rar
```

The above script divides data into 20% test set and 80% training set.

Training Text Classification Model and Predicting Sentiment

We have divided our data into training and testing set. Now is the time to see the real action. We will use the Random Forest Algorithm to train our model. You can you use any other model of your choice.

To train our machine learning model using the random forest algorithm we will use `RandomForestClassifier` class from the `sklearn.ensemble` library. The `fit` method of this class is used to train the algorithm. We need to pass the training data and training target sets to this method. Take a look at the following script:

```
classifier = RandomForestClassifier(n_estimators=1000, random_state=0)
classifier.fit(X_train, y_train)
```

Finally, to predict the sentiment for the documents in our test set we can use the `predict` method of the `RandomForestClassifier` class as shown below:

```
y_pred = classifier.predict(X_test)
```

Congratulations, you have successfully trained your first text classification model and have made some predictions. Now is the time to see the performance of the model that you just created.

Evaluating the Model

To evaluate the performance of a classification model such as the one that we just trained, we can use metrics such as the confusion matrix, F1 measure, and the accuracy.

To find these values, we can use `classification_report`, `confusion_matrix`, and `accuracy_score` utilities from the `sklearn.metrics` library. Execute the following script to do so:

```
from sklearn.metrics import classification_report, confusion_matrix, accurac

print(confusion_matrix(y_test,y_pred))
```

```
print(classification_report(y_test,y_pred))
print(accuracy_score(y_test, y_pred))
```

The output looks like this:

```
[[180  28]
 [ 30 162]]
             precision    recall  f1-score   support

          0       0.86      0.87      0.86       208
          1       0.85      0.84      0.85       192

avg / total       0.85      0.85      0.85       400

0.855
```

From the output, it can be seen that our model achieved an accuracy of 85.5%, which is very good given the fact that we randomly chose all the parameters for `CountVectorizer` as well as for our random forest algorithm.

Saving and Loading the Model

In the script above, our machine learning model did not take much time to execute. One of the reasons for the quick training time is the fact that we had a relatively smaller training set. We had 2000 documents, of which we used 80% (1600) for training. However, in real-world scenarios, there can be millions of documents. In such cases, it can take hours or

even days (if you have slower machines) to train the algorithms. Therefore, it is recommended to save the model once it is trained.

We can save our model as a `pickle` object in Python. To do so, execute the following script:

```
with open('text_classifier', 'wb') as picklefile:
    pickle.dump(classifier,picklefile)
```

Once you execute the above script, you can see the `text_classifier` file in your working directory. We have saved our trained model and we can use it later for directly making predictions, without training.

To load the model, we can use the following code:

```
with open('text_classifier', 'rb') as training_model:
    model = pickle.load(training_model)
```

We loaded our trained model and stored it in the `model` variable. Let's predict the sentiment for the test set using our loaded model and see if we can get the same results. Execute the following script:

```
y_pred2 = model.predict(X_test)

print(confusion_matrix(y_test, y_pred2))
print(classification_report(y_test, y_pred2))
print(accuracy_score(y_test, y_pred2))
```

The output looks like this:

```
[[180  28]
 [ 30 162]]
             precision    recall  f1-score   support

          0       0.86      0.87      0.86       208
          1       0.85      0.84      0.85       192

avg / total       0.85      0.85      0.85       400
```

```
0.855
```

The output is similar to the one we got earlier which showed that we successfully saved and loaded the model.

Going Further - Hand-Held End-to-End Project

Your inquisitive nature makes you want to go further? We recommend checking out our ***Guided Project***: *"Image Captioning with CNNs and Transformers with Keras"*.

> *In this guided project - you'll learn how to build an image captioning model, which accepts an image as input and produces a textual caption as the output.*

You'll learn how to:

- Preprocess text
- Vectorize text input easily
- Work with the `tf.data` API and build performant Datasets
- Build Transformers from scratch with TensorFlow/Keras and KerasNLP - the official horizontal addition to Keras for building state-of-the-art NLP models
- Build hybrid architectures where the output of one network is encoded for another

How do we frame image captioning? Most consider it an example of generative deep learning, because we're teaching a network to generate descriptions. However, I like to look at it as an instance of neural machine translation - we're translating the visual features of an image into words. Through translation, we're generating a new representation of that image, rather than just generating new meaning. Viewing it as translation, and only by extension generation, scopes the task in a different light, and makes it a bit more intuitive.

Framing the problem as one of translation makes it easier to figure out which architecture we'll want to use. Encoder-only Transformers are great at understanding text (sentiment analysis, classification, etc.) because Encoders encode meaningful representations. Decoder-only models are great for generation (such as GPT-3), since decoders are able to infer meaningful representations into another sequence with the same meaning. **Translation is typically done by an encoder-decoder architecture**, where encoders encode a meaningful representation of a sentence (or image, in our case) and decoders learn to turn this sequence into another meaningful representation that's more interpretable for us (such as a sentence).

Conclusion

Text classification is one of the most commonly used NLP tasks. In this article, we saw a simple example of how text classification can be performed in Python. We performed the sentimental analysis of movie reviews.

I would advise you to change some other machine learning algorithm to see if you can improve the performance. Also, try to change the parameters of the `CountVectorizer` class to see if you can get any improvement.

#python #machine learning #scikit-learn #nlp

Last Updated: July 21st, 2022

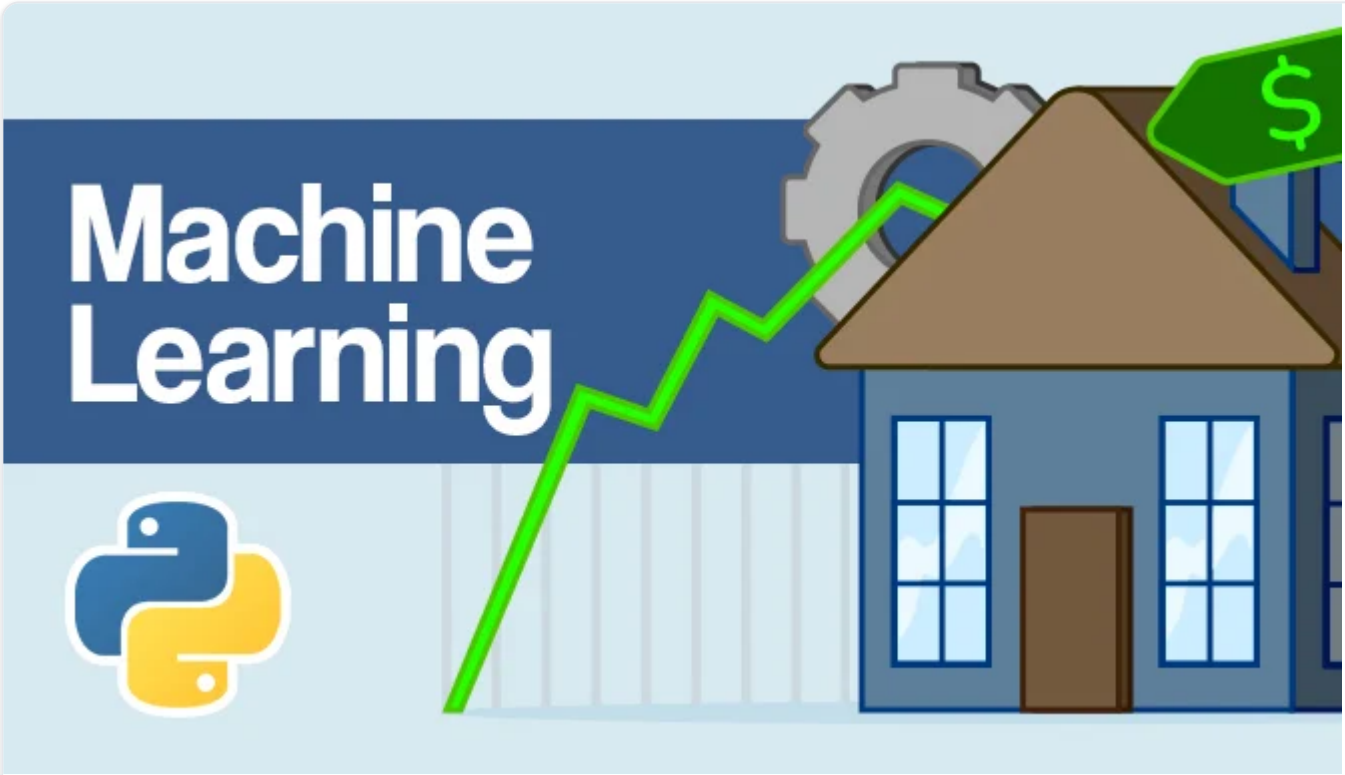
Was this article helpful? ☆☆☆☆☆

You might also like...

- Scikit-Learn's train_test_split() - Training, Testing and Validation Sets
- Simple NLP in Python with TextBlob: N-Grams Detection
- Dimensionality Reduction in Python with Scikit-Learn

Usman Malik *Author*

Programmer | Blogger | Data Science Enthusiast | PhD To Be | Arsenal FC for Life



Project

Hands-On House Price Prediction - Machine Learning in Python

#deep learning #tensorflow #machine learning #python

If you've gone through the experience of moving to a new house or apartment - you probably remember the stressful experience of choosing a property,...

Details →

Project

Image Captioning with CNNs and Transformers with Keras

#artificial intelligence #deep learning #python #nlp

In 1974, Ray Kurzweil's company developed the "Kurzweil Reading Machine" - an omni-font OCR machine used to read text out loud. This machine...

David Landup

Details →