NLP Part 2| Pre-Processing Text Data Using Python

Let's prep/clean our data for analysis and do not forget this is an iterative process.

Kamil Mysiak Apr 19, 2019 · 7 min read ★

by Kamil Mysiak

Photo by Dmitry Ratushny on Unsplash

My previous **article** explored the concept of scraping textual information from websites using a python library named BeautifulSoup. We were quickly able to scrape employee company ratings from Indeed.com and export the data to a local CSV file. Scraping the data is merely the first step in gleaning useful insights from our newly acquired text data. The purpose of this article is to take that next step and apply a few standard pre-processing steps in order to prep the data for analysis.

Which pre-processing methods you choose to perform will depend on your data, desired outcome and/or how you choose to analyze your data. That said, the pre-processing methods listed below are some of the most common methods typically being utilized.

1. Importing Libraries along with our Data
2. Expanding Contractions
3. Language Detection
4. Tokenization
5. Converting all Characters to Lowercase
6. Removing Punctuations
7. Removing Stopwords
8. Parts of Speech Tagging
9. Lemmatization

Importing the Necessary Libraries

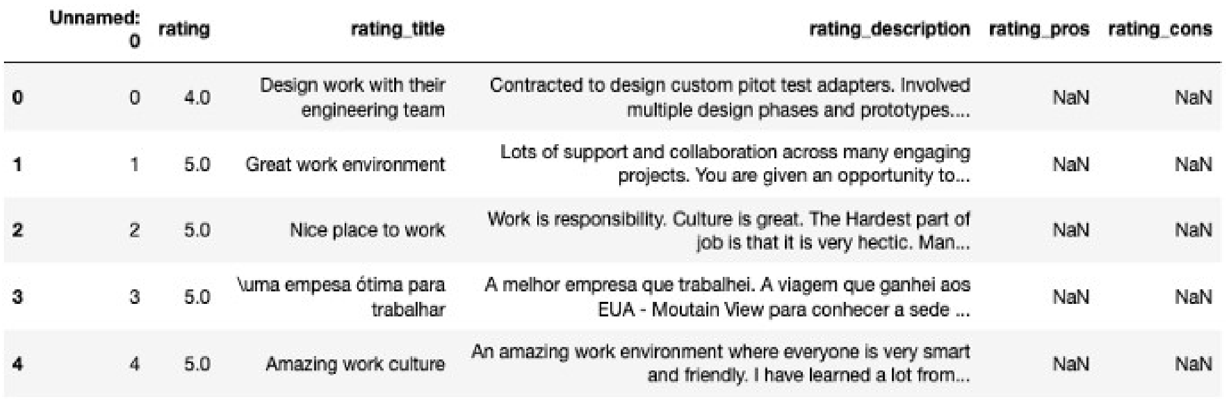
```
import pandas as pd
import numpy as np
import nltk
import string
import fasttext
import contractions
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords, wordnet
from nltk.stem import WordNetLemmatizer
```

```
plt.xticks(rotation=70)
pd.options.mode.chained_assignment = None
pd.set_option('display.max_colwidth', 100)
%matplotlib inline
```

Importing our Data

We are going to import the scraped employee review ratings obtained in our previous **tutorial** and quickly examine the data.

```
with open('indeed_scrape.csv') as f:
    df = pd.read_csv(f)
f.close()
```



	Unnamed: 0	rating	rating_title	rating_description	rating_pros	rating_cons
0	0	4.0	Design work with their engineering team	Contracted to design custom pitot test adapters. Involved multiple design phases and prototypes....	NaN	NaN
1	1	5.0	Great work environment	Lots of support and collaboration across many engaging projects. You are given an opportunity to...	NaN	NaN
2	2	5.0	Nice place to work	Work is responsibility. Culture is great. The Hardest part of job is that it is very hectic. Man...	NaN	NaN
3	3	5.0	\uma empesa ótima para trabalhar	A melhor empresa que trabalhei. A viagem que ganhei aos EUA - Moutain View para conhecer a sede ...	NaN	NaN
4	4	5.0	Amazing work culture	An amazing work environment where everyone is very smart and friendly. I have learned a lot from...	NaN	NaN

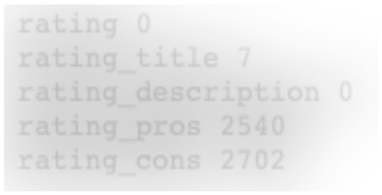
We will be focusing our attention on the "rating" and "rating_description" columns as they contains the most valuable qualitative information. Although we wouldn't be applying any pre-processing steps to the "rating" column.

First, let's drop the "Unnamed: 0" column as it simply duplicates the index.

```
df.drop('Unnamed: 0', axis=1, inplace=True)
```

Next, let's examine if we have any missing values. It seems both "rating" and "rating_description" do not contain any missing values.

```
for col in df.columns:
    print(col, df[col].isnull().sum())
```



```
rating 0
rating_title 7
rating_description 0
rating_pros 2540
rating_cons 2702
```

```
rws = df.loc[:, ['rating', 'rating_description']]
```

Text Pre-Processing

Expanding Contractions

Contractions are those little literary shortcuts we take where instead of "Should have" we prefer "Should've" or where "Do not" quickly becomes "Don't". We are going to add a new column to our dataframe called "no_contract" and apply a lambda function to the "rating_description" field which will expand any contractions. Be aware of the fact the expanded contractions will be effectively tokenized together. In other words, "I've" = "I have" instead of "I", "have".

```
rws['no_contract'] = rws['rating_description'].apply(lambda x:
[contractions.fix(word) for word in x.split()])
rws.head()
```

We ultimately would want the expanded contractions to be tokenized separately into "I", "have", therefore, let's convert the lists under the "no_contract" column back into strings.

```
rws['rating_description_str'] = [' '.join(map(str, l)) for l in
rws['no_contract']]
```

```
rws.head()
```

English Language Detection

The next step is to identify the language in which each review is written in and then remove any non-English reviews. We first have to download the pre-trained language model for our fasttext library (*everyone needs to thank facebook for this one*). TextBlob is a common library used to detect string language but it will quickly throw you an error when you are parsing a large amount of text. Once we have downloaded the model, we'll use a for-loop to iterate through our reviews. The result is a tuple of the predicted language and the probability of the prediction. In our case, we only require the first (ie. language prediction) portion of the tuple. Finally, we select only the last two characters.

```
pretrained_model = "lid.176.bin"
model = fasttext.load_model(pretrained_model)

langs = []
for sent in rws['rating_description_str']:
    lang = model.predict(sent)[0]
    langs.append(str(lang)[11:13])

rws['langs'] = langs
```

Now all we have to do is remove any non-english reviews.

Tokenization

Now that we have removed any non-English reviews let's apply our tokenizer in order to split each individual word into a token. We will apply NLTK.word_tokenize() function to the “rating_description_str” column and create a new column named “tokenized”.

```
rws['tokenized'] =
rws['rating_description_str'].apply(word_tokenize)
rws.head()
```

Converting all Characters to Lowercase

Transforming all words to lowercase is also a very common pre-processing step. In this case, we will once again append a new column named “lower” to the dataframe which will transform all the tokenized words into lowercase. However, because we have to iterate over multiple words we will use a simple for-loop within a lambda function to apply the “lower” function to each word.

```
rws['lower'] = rws['tokenized'].apply(lambda x: [word.lower()
for word in x])
rws.head()
```

Removing Punctuations

Punctuation is often removed from our corpus since they serve little value once we begin to analyze our data. Continuing the previous pattern, we will create a new column which has the punctuation removed. We will again utilize a for-loop within a lambda function to iterate over the tokens but this time using an IF condition to only output alpha characters. It might be a little difficult to see but the tokenized "period" in the " lower" column has been removed.

```
punc = string.punctuation
rws['no_punc'] = rws['lower'].apply(lambda x: [word for word in
x if word not in punc])
rws.head()
```

Removing Stopwords

Stopwords are typically useless words and do not add much meaning to a sentence. In the English language common stopwords include "you, he, she, in, a, has, are, etc.". First, we need to import the NLTK stopwords library and set our stopwords to "english". We are going to add a new column "no_stopwords" which will remove the stopwords from the "no_punc" column since it has been tokenized, had been converted to lowercase and punctuation was removed. Once again a for-loop within a lambda function will iterate over the tokens in "no_punc" and only return the tokens which do not exist in our "stop_words" variable.

```
stop_words = set(stopwords.words('english'))
rws['stopwords_removed'] = rws['no_punc'].apply(lambda x: [word
for word in x if word not in stop_words])
rws.head()
```

Stemming vs Lemmatization

The idea of stemming is to reduce different forms of word usage into its root word. For example, “drive”, “drove”, “driving”, “driven”, “driver” are derivatives of the word “drive” and very often researchers want to remove this variability from their corpus. Compared to lemmatization, stemming is certainly the less complicated method but it often does not produce a dictionary-specific morphological root of the word. In other words, stemming the word “pies” will often produce a root of “pi” whereas lemmatization will find the morphological root of “pie”.

Instead of taking the easy way out with stemming, let’s apply lemmatization to our data but it requires some additional steps compared to stemming.

First, we have to apply parts of speech tags, in other words, determine the part of speech (ie. noun, verb, adverb, etc.) for each word.

```
rws['pos_tags'] =
rws['stopwords_removed'].apply(nltk.tag.pos_tag)
rws.head()
```

We are going to be using NLTK's word lemmatizer which needs the parts of speech tags to be converted to wordnet's format. We'll write a function which make the proper conversion and then use the function within a list comprehension to apply the conversion. Finally, we apply NLTK's word lemmatizer.

```
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
```

```
rws['wordnet_pos'] = rws['pos_tags'].apply(lambda x: [(word,
get_wordnet_pos(pos_tag)) for (word, pos_tag) in x])
rws.head()
```

Now we can apply NLTK's word lemmatizer within our trusty list comprehension. Notice, the lemmatizer function requires two parameters the word and its tag (in wordnet form).

```
wnl = WordNetLemmatizer()
rws['lemmatized'] = rws['wordnet_pos'].apply(lambda x:
[wnl.lemmatize(word, tag) for word, tag in x])
rws.head()
```

Get this newsletter

exploratory data analysis which you can read all about in my next **blog** .

Data Science Nltk Text Preprocessing

```
rws.to_csv('indeed_scrape_clean.csv')
```

Text pre-processing can quickly become a rabbit-hole of edits and further techniques but we have to draw a line somewhere. I often will go back to the data pre-processing stage as I move further along in my analysis process because I've uncovered some data issues which need to be addressed. That said, learn to draw a line somewhere.