Published in Towards Data Science

Lisa A. Chalaguine Follow

Jul 13, 2020 · 7 min read · Listen

Getting started with text analysis in Python

A pragmatic step-by-step tutorial for data analysts who are stuck with Excel for text analysis

sentiment analysis. However, I honestly do not know why someone would do that if free and less awkward tools exist — like Python.

I have outlined in a previous article, that many people are reluctant to pick up coding because they believe that it is difficult and in-depth math knowledge is required. Neither is true. If you can write long and awkward functions in Excel, let me reassure you that Python is way easier and more intuitive. Besides, there exist various Python libraries for natural language proce[88] a huge plethora of in-built functions that will do the heavy lifting for you.

In this article, I want to start with the very basics of text analysis in Python. Some Python knowledge is necessary, so I suggest you check out my previous article in which I give tips on how to get started with Python or R for Data Analysis. This article will be of a similar format. There are plenty of tutorials and articles on how to get started with NLP in Python, some of which I will link to in this article. However, I want to give a pragmatic example on how to deal with real-world text data which you might encounter in your daily work life. I want to show how to apply some simple, but powerful, text analysis techniques and how to tackle problems you might run into.

I will give some code samples from a notebook that I created for a friend who wanted to get started with text analysis in Python for his job. The whole notebook is available here, if you want to go through it. It is not the most exciting dataset since I tried to find a public dataset which is similar to the type of data that he has to analyse at work. The dataset contains US railroad incidents from 2019.

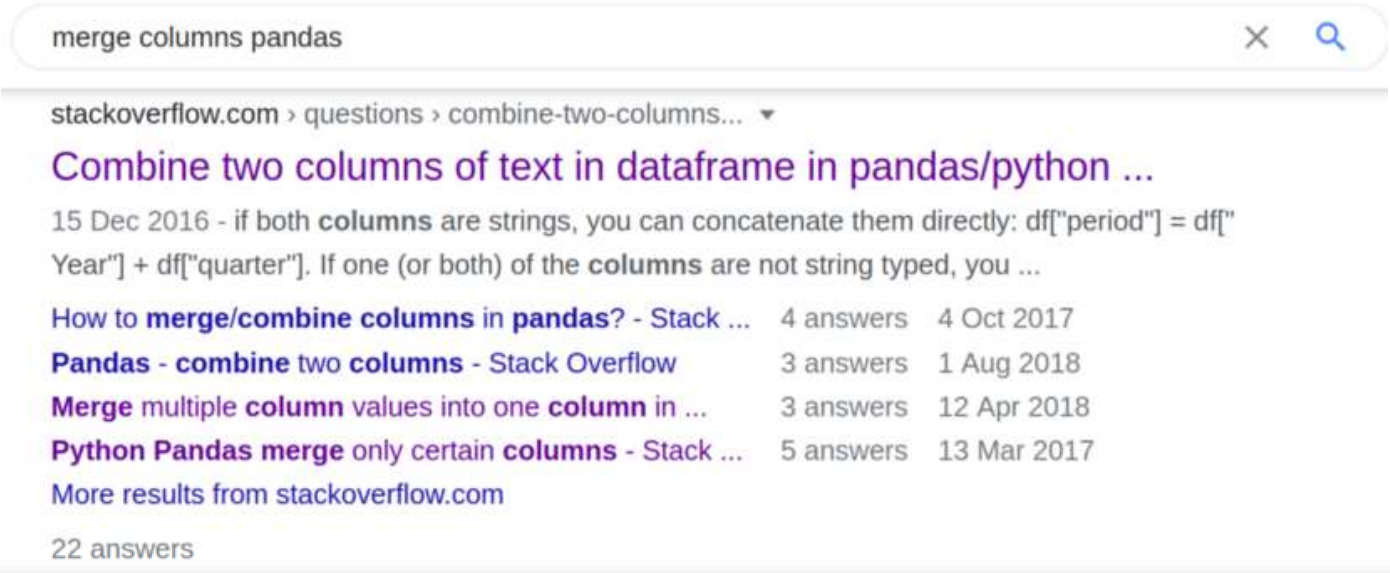
Asking the right question

Unlike the nicely cleaned and formatted datasets that you get to analyse in tutorials, real-world data is often messy and noisy. For example, when importing the US railroad data, the incident text which was supposed to be in one column, somehow ended up in 15 columns. I am not sure about MS Excel (haven't used that in years) but in LibreOffice Calc (the open-source version that comes with Ubuntu) you'd have to merge the columns row-by-row. This is already tedious for 20 rows, not to mention for

In my opinion, the best way to learn something new, is to get your hands dirty. Of course, you should spend some time to familiarise yourself with the basics, but no one has time to do tutorials on all useful Python libraries. If you don't know how to perform a certain operation (like merging columns using *pandas*) – Google (or Baidu, or Yandex…) is your friend. I can assure you that someone has asked that question before and got an answer on *stackoverflow*.

The hardest part for everyone new to programming (or anything actually) is to formulate the right question when they get stuck, in order to find the answer they are looking for. Asking the right question and googling is a skill and comes with practice. In the beginning, it might, therefore, take you longer to find an answer on the web. But don't worry — you will pick up the necessary vocabulary and experience soon enough.

> ***Problem:*** *Data which should be in one column is split into multiple columns*
> ***End goal:*** *Have one new column which contains all the data from the columns and delete the redundant columns*
>
> ***Solution:*** *Merge columns*
>
> ***Google query:*** *Merge multiple columns pandas*



columns.

```
df['ColumnA'] = df[df.columns[1:]].apply(
  lambda x: ' '.join(x.dropna().astype(str)),
  axis=1
)
```

mergecols.py hosted with ❤ by GitHub view raw

This is a very neat solution. Why? Because it's general and takes into consideration other problems you could encounter down the line: like NaN-values (empty cells) and different data types in different columns (NaNs are of type *float*). I am going through step-by-step what the function does in my dummy notebook.

Non-Textual Preprocessing

Given I want to focus on text data, I will not cover how to handle missing data. I am giving some examples of how to impute missing numerical data in this notebook (note: It's in R, not Python). For the railroad incident dataset, I just checked for duplicate rows and dropped them. I suggest you do everything necessary with your data that makes your life easier. If you are only interested in certain columns — select only those and store them in a data frame. If you are interested only in a certain part of your data — select that (e.g. only incidents that happen in Kansas City).

Text Preprocessing

In this section I want to go over some important NLP concepts and show code examples on how to apply them on text data.

Stopword removal

Stop words are words that may not carry any valuable information, like articles (*"the"*), conjunctions (*"and"*), or propositions (*"with"*). Why would you want to remove them? Because finding out that *"the"* and *"a"* are the most common words in your dataset doesn't tell you much about the data. However, if the most common words include

The NLTK stopword list, however, only has around 200 stopwords. The stopword list which I use for my text analysis contains almost 600 words[1]. I keep it in a file, where I have deleted apostrophes (*ain't → aint*) because I remove punctuation and tokenise the text before I remove stopwords.


negative reviews: "*I will **not** buy this product again. I saw **no** benefits in using it*". I am addressing negation in the notebook.

Tokanisation into n-grams

As you have already seen in the code snippet above, we need to tokenise the text into single words and store them in a list, in order to loop through the list and remove the stop words. Single words are called unigrams, two words bi-grams, and three words tri-grams.

When dealing with a new dataset I often find it helpful to extract the most common words to get an idea of what the data is about. You usually want to extract the most common unigrams first, but it can also be useful to extract n-grams with larger n to identify patterns. You should, however, still read at least a few of your data points in order to identify potential problems. In the railroad incident texts, for example, I have noticed that some reports specify that *no* damage or hazmat (hazardous materials) release occurred. I then used bi-grams and tri-grams to identify negations and distinguish between incidents where damage and hazmat release occurred and those where it did not.

NLTK has in-built bigrams, trigrams and ngrams functions.


Punctuation Removal with Regular Expressions

Punctuation is not always useful in predicting the meaning of texts, so they are often removed along with stop words. Removing punctuation leaves you only with alpha-numeric characters. I will not get into Regex expressions here — there are many tutorials on them on the internet. Below are two regex expressions — one which deletes all punctuation and numbers, and one that leaves numbers in the text.

Note that whether deleting punctuation is a good idea depends entirely on your data. If you want to extract specific information, e.g. the $ amount of the damages, you want to keep the dollar sign in so you can extract the amount that follows it. Another example, where keeping punctuation is beneficial is for training a spam-ham classifier. Spam e-mails or texts usually contain more punctuation and can, therefore, be a useful feature.

Lemmatisation and Stemming

Lemmatisation and stemming both refer to a process of reducing a word to its root. The difference is that **stem** might not be an actual word whereas, a lemma is an actual word. It's a handy tool if you want to avoid treating different forms of the same word as different words, e.g. *derailment, derailing, derail* and *derailed*.

> ***Lemmatising:*** *considered, considering, consider → "consider"*
> ***Stemming:*** *considered, considering, consider → "consid"*

I personally have never noticed a significat difference between lemmatising and stemming when training classifiers. However, I suggest you try out yourself. NLTK comes with many different in-built lemmatisers and stemmers, so just plug and play.


Putting it all together

Now you should know enough to get you started with text analysis in Python. Let's put it all together by using a dummy dataset I created with 5 movie reviews. The notebook can be found for download here.

Jupyter Notebook to demonstrate all concepts covered in the article

The preprocessing function in the notebook I have written at the very beginning of my PhD and have been copy-pasting it throughout different projects. If you always deal with the same type of data in your job, it's easy to code up some functions that automate the preprocessing, exploration, and analysis, which you can copy-paste every time you have to deal with a new dataset.

I hope you found this article useful and it will help you getting started with Python pandas, NLTK, and any other libraries out there that will make your job easier. I will cover some more advanced NLP topics in the next article, so if you have any suggestions on what I should cover, let me know!

[1] Lewis, David D., et al. (2004) "Rcv1: A new benchmark collection for text categorization research." *Journal of machine learning research* 5: 361–397.