PharmaSUG 2019 - Paper AD-129

Automate the Mundane: Using Python for Text Mining

Nathan Hammett Kosiba, Rho Inc.

ABSTRACT

As programmers and biostatisticians, we have a number of tasks that are broken down to a "copy and paste from document A to document B" scenario. These tasks range from copying the name of the study and sponsor into a SAP to pulling Inclusion and Exclusion criteria for the TI domain. We perform these tasks ad nauseam. While SAS is very good at dealing with structured data and analyses, it is not well equipped to deal with unstructured text such as a protocol. Python, on the other hand, has numerous tools for dealing with unstructured text and breaking it into meaningful pieces of information. While Python can do a vast amount of different things, we will be focusing on how to process information in a protocol to populate Metadata and create Trial Design datasets. Using Python and regular expressions, we can process large amounts of information held in a protocol or other document and retrieve what we need without much manual effort. These tools are often written with user interfaces included so there is no programming knowledge required for the end-user. In addition, Python integrates with a variety of existing systems and/or produces output that is easily used as input for these systems. This flexibility supports nearly fully automated results. This presentation will explore some of the Python packages that facilitate this process and how Python can be used to automate mundane tasks.

INTRODUCTION

With the emergence of Python as a popular programming language in the clinical trials industry, we must look for how best to utilize its power and flexibility. While it can create stunning data visual displays and run well-structured analyses, I believe that some of the more abstract cases are where it can help us the most. One such case is text mining, exploring and analyzing large amounts of unstructured text data to identify trends, patterns, concepts and extract important information. As programmers and biostatisticians, we use a number of documents that contain important information that we need to do our work. We end up copying numerous pieces of text from one document to another. While this process may seem easy, if you are spending this time on every study the minutes and hours start to add up. With Python, these documents and the text contained within are easily analyzed and extracted and even used to create pieces of other documents. In this paper, we will explore text mining as a concept and how to utilize some of the packages that have been purpose-built for this application in Python. The case-study used in this paper is the extraction of Inclusion and Exclusion criteria from a Study Protocol to be used as metadata for a study or to create the Trial Inclusion SDTM domain. We will start with a quick overview of Python and why it works well for this application

WHY PYTHON?

Python is an open source, general use, module based programming language. Python is used for everything from general data manipulation to machine learning applications and building web interfaces. The base installation of Python offers powerful tools for basic programming, but there are many third party libraries that can be added to Python to further enhance its capabilities. In addition, once a programmer is comfortable with programming in Python they can create their own packages tailored to the task at hand.

One useful feature of Python is the ability to process many types of documents and interface with a multitude of different systems. Python has the ability to interpret different file types including PDF and Microsoft Word files. These types of files are problematic for SAS to fully process in a meaningful way. Python can use regular expressions to conduct pattern searches as well as parse the text into individual pieces. More complicated text analyses such as Natural Language Processing are also applied using Python. These methods help us make full use of the different types of data that we have and accept the different forms that it comes in. In this paper, we will specifically be looking at PDF documents and how text and other objects are extracted from them and analyzed.

Python is also used to write user interfaces to create tools that are available to a wider audience. While SAS macros execute a set of commands given different parameters, user interfaces in Python are used to give non-programmers the ability to use tools without any coding knowledge needed. This broadens the audience for Python applications and allows the wide use and distribution of them within a company, if done correctly. Although Python programs can be run using the command line and global variables (similar to SAS), user interfaces greatly enhance the ease of use of any program.

In Python, the basic package used to create a local interface is called Tkinter. While there are a plethora of other packages that create interfaces, Tkinter is the building block of most local user interfaces in Python and the one I believe offers the user the most options. Another way to create interfaces in Python is through web applications. These are built through a package such as Flask and deployed through a server. While not necessary, building a user interface helps users navigate the inputs to a given application. In addition, the programmer(s) can add help text to explain limitations and give user instructions so deployment goes smoothly. Below is a short example of how to create a simple interface in Python that can accept a file path.

```
from tkinter import *

# setup user input window
interface = Tk()
interface.title("Tkinter Example")

#define input paths
labeltxtpath=StringVar()
labeltxtpath.set("Input Path: ")
labelDirs=Label(interface, textvariable=labeltxtpath)
labelDirs.grid(row=1,column=0)

txt = StringVar()
txtpath = Entry(interface, textvariable=intxt, width=75)
txtpath.grid(row=1,column=1)
txtpath.insert(0,"Enter Input Path Here")

interface.mainloop()
```

This code will return the user input file path as a string variable that is used later in the program. While this is a basic user interface, with a broader knowledge of Tkinter, intricate ones are created that let the user browse to a file path for input. While user interfaces are not the most exciting thing to program, they are essential to deployment of any application.

TEXT MINING

WHAT IS IT?

Text mining is exploring and analyzing large amounts of unstructured text data to identify trends, patterns, concepts and extract important information. It is implemented in a number of different ways, from searching a document and extracting information, to creating text visuals such as word clouds to using Natural Language Processing for analysis and creation of text. All of these methods help us better understand the documents we look at on a daily basis. We can use text mining to automate the mundane tasks we do. In this paper, we will focus mainly on the analysis of text and the extraction of pieces of text for creating or updating metadata or other documents. We will use the example of collecting Inclusion and Exclusion criteria contained in the Study Protocol for use as metadata or in the Trial Inclusion SDTM domain.

USING PYTHON TO MINE TEXT

There are a few pieces to consider when doing any sort of data analysis. First we consider the location of our data and how to best access it. In our case, this access point comes in the form of a document processor in Python that extracts text and other objects from the document. In this paper, we look at extracting and mining text from a PDF document using the Python package PDFMiner. Shown below is some basic PDFMiner code that is used to extract text off a page and store the text in a list. In addition, the package RE (regular expression) is used to split the text into pages.

```
from pdfminer.converter import TextConverter
from pdfminer.pdfinterp import PDFPageInterpreter
from pdfminer.pdfinterp import PDFResourceManager
from pdfminer.pdfpage import PDFPage
from pdfminer.pdfparser import PDFParser
from pdfminer.pdfdocument import PDFDocument

# PDF text conversion
def extract_text_by_page(pdf_path):
    with open(pdf_path, 'rb') as fh:
          for page in PDFPage.get_pages(fh,
                                        caching=True,
                                        check_extractable=True):
                resource_manager = PDFResourceManager()
                handled = io.StringIO()
                converter = TextConverter(resource_manager, handled)
                page_interpreter = PDFPageInterpreter(resource_manager,
                                      converter)
                page_interpreter.process_page(page)

                text = handled.getvalue()
                yield text

                # close open handles
                converter.close()
                handled.close()

# create list with page text as elements
import re

pagelist=[]
for page in extract_text_by_page(pdf_path):
pagelist.append([re.sub(r'\s+',' ',page)])
```

Once the text is extracted from the page, the user must store it in an object. Usually text is stored in a long string or in a list with each element being the text from a single page. With free text isolated, we can now analyze it for trends and parse out useful information. In this case, we are trying to find the Inclusion and Exclusion criteria in the protocol and break them up into the individual criteria. The table of contents is a huge help in this part by telling use which pages the Inclusion and Exclusion criteria are on. This gives us a smaller amount of text to look through and as long as the text is stored in a list the text corresponding to a given page is easily identified.

When searching for certain pieces of text, it is helpful to identify how that text would normally be structured. For example, Inclusion and Exclusion criteria will usually be in a numbered list. Therefore, when parsing free text to identify the individual criteria we look for sections that begin with a number and possibly some additional text pieces. Below is an example of some Inclusion and Exclusion criteria that could be found in a protocol.

4.1 Inclusion Criteria

Subjects will be allowed to participate in this study only if they meet all of the following criteria:

1. Male or Female subjects 18 to 40 years of age, inclusive, at the time of informed consent
2. Subjects who can comply with the rules of concomitant medications and therapies usage as defined in the protocol.

4.2 Exclusion Criteria

Subjects will not be allowed to participate in this study if they meet any of the following criteria:

1. Subjects who are pregnant or breast-feeding, or women of childbearing potential with a positive serum or urine pregnancy test(s).
2. Subjects with medical conditions and/or diseases that the investigator believes could affect the study results or interfere with safe conduct of the study.

To systematically search and parse text based on conditions in Python, regular expressions are often used. These are similar to the Perl regular expressions that are used in SAS. When creating a pattern to look for, you need to take into account what else is included in surrounding text. For example, in this case we don't just want to look for a number followed by words to parse the criteria. There could be numeric lab values or age ranges that would match the pattern and cause the program to misfire. The regular expression package in python (Regex) can use elements corresponding to characters, numbers, and special characters to create a pattern to use when parsing the text. In the case above, the pattern used is as follows: one period (\.), one space, one number (\d), possibly another number (?\d), one period (\.), and one space. The combined pattern used to match is (\. \d?\d\.), which successfully parsed the criteria out of free text and into a list. The basic code used is shown below

```
# parse criteria
import re

criteria =re.split('(\. \d?\d\. )' , pagestring)
```

The first piece uses the split function to split the string variable that contains the page text into a list with each individual numbered criterion as an element. While the pattern used for matching will need to be customized for different applications, it does not need to be overly complicated. With a basic string the Inclusion and Exclusion criteria were split into their pieces. In some cases, the pattern is set in the user interface depending on the text involved.

After finding the criteria, the user has total control over what to do with the data. If the user wishes to output it as an Excel file, they can use the Pandas package to create a dataframe and then write that to an Excel spreadsheet. If the user needs to upload this to an internal oracle database, CX_Oracle is a package that enables the user to interact with oracle databases.

CONCLUSION

While Python has a lot of different applications within the framework of clinical trials, I believe that text mining at all levels has dramatic impacts on the efficiency and accuracy of our data analysis. As Python becomes more prevalent in the industry I envision many similar applications developed to eliminate tedious and mundane processes. Even just in the realm of text mining, there is a huge amount of other information that can be programmatically extracted from a protocol. One of the best things about Python is that the user does not need to know all the nuts and bolts to complete basic tasks. In this case, I used a small number of packages to complete this project. While this is not a complicated text mining endeavor, the time it saves is worth the time put in to learn the packages and write and test the code. In the case of text mining, a next logical step from creating applications based on pattern searching is to look into Natural Language Processing and identify new use cases. I look forward to seeing what our colleagues create as we utilize the power of Python more.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Nathan Hammett Kosiba
Rho, Inc.
Nathan_Kosiba@rhoworld.com