Paper 2861-2018

At Your Service: Using SAS® Viya™ and Python to Create Worker Programs for Real-Time Analytics

Jon Klopfer, Scott Koval, and Mia List, Pinnacle Solutions, Inc.

ABSTRACT

Being able to automatically analyze and score data on demand has always been a challenge faced when creating a service program. SAS Viya brings with it new and exciting ways to help tackle this task. When combining SAS programming with the Python language, it is possible to create worker programs that can be deployed to automatically process any incoming data. This paper provides ideas in how to create an intelligent worker program that will wake up when needed, analyze data, and then rest until it is needed again. Utilizing SAS Viya programming is essential for creating any real-time analytics environment and can be integrated to suit any business need.

INTRODUCTION

The need for real-time analytics has been growing. In today's world, data flows continuously across the internet. Savvy companies are able to create programs that work around the clock to capture, process, and make sense of this data. These worker programs run constantly and ideally have a single purpose. Once this task is complete, the worker program waits patiently until it is once again needed. Groups of worker programs can exist on a platform in a cloud computing environment, intelligently delivering data to one another for processing purposes. This low level file exchange process allow for platform independent communication that can run on any environment.

The SAS Viya platform offers a wide variety of data mining and machine learning solutions. This paper provides an example of applying the power of SAS Viya analytics to an existing Amazon Web Services (AWS) system without disrupting the already developed framework. The SAS Viya platform is able to run continuously and seamlessly alongside other programs.

SAS Viya utilizes a SAS Cloud Analytic Services (CAS) server, which is the main component of the infrastructure. The CAS server not only allows SAS applications to communicate with it, but also provides access through other programming languages, such as Python, R, Lua, and Java. The SAS Scripting Wrapper for Analytics Transfer (SWAT) package allows programmers to access and send commands to the CAS server using their preferred language of choice.

Figure 1 is an example of how SAS Viya was successfully integrated to run alongside an existing codebase. SAS Viya and the SWAT package allowed developers to use both SAS analytics and Python to further develop a real time service offering. In this instance, social media data is being collected on various subjects. A text mining service provided by SAS was added to this process in order to classify user comments.

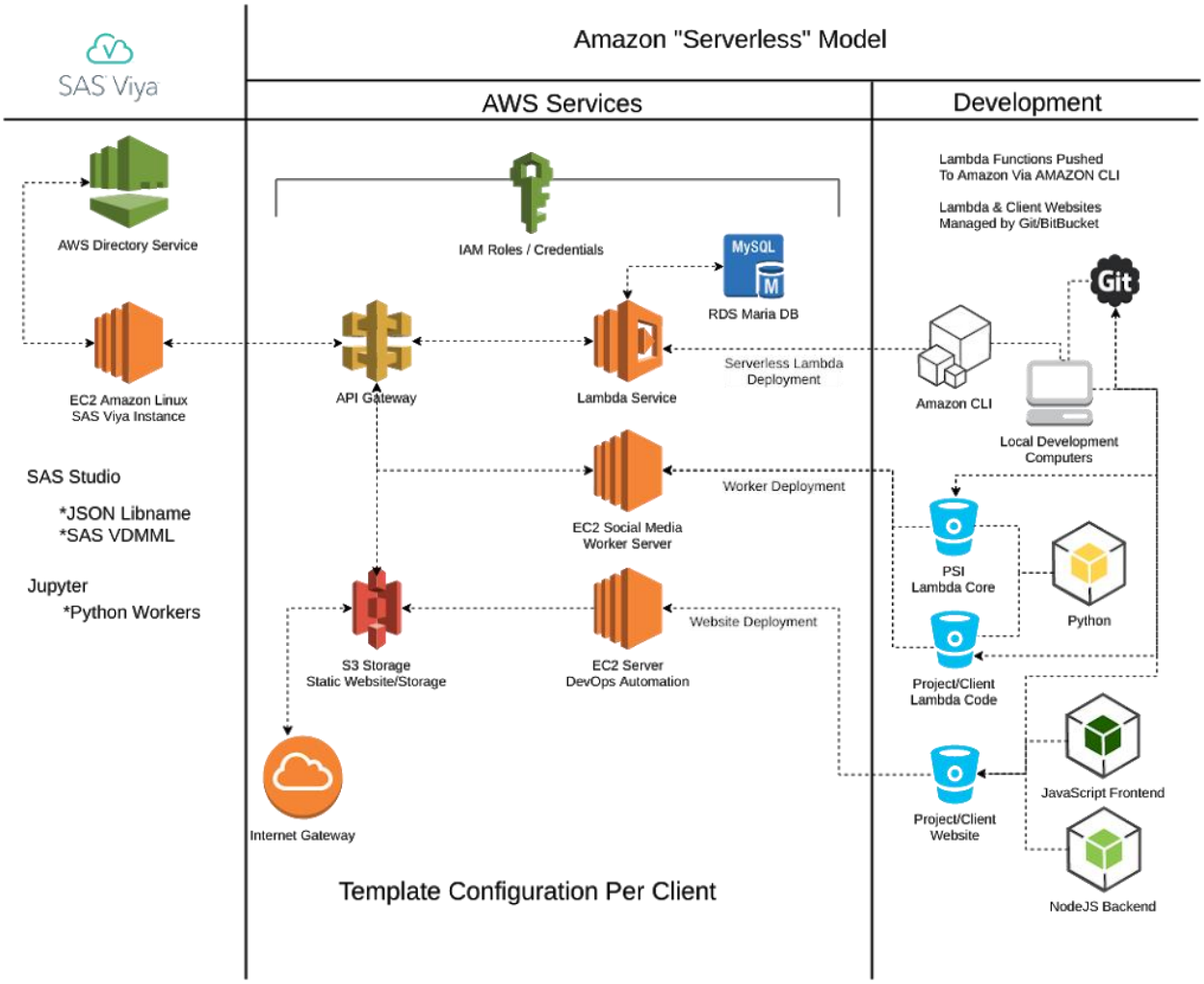

Figure 1. Architecture Diagram of Server

EXAMPLE APPLICATION

Social media and online interactions have grown exponentially over the last few decades. Many companies rely on these platforms to allow people to share comments, opinions, and feedback; however, anonymity, asynchronous communication, and a lack of empathy can lead to an increase in negativity towards one another (Kaggle, 2017). This type of behavior can stifle expression and breed hostile actions between the users. There is a growing concern to identify toxic, obscene, or hateful behavior on social media websites and filter it out.

In this example, an already established process exists to collect and report on social media content. An AWS system is in production that is utilizing social media APIs for data collection and storage. A SAS Viya program was developed using machine learning to text mine and classify these comments. Python is then able to use this model in a worker program to classify new data and export it into a file that is then delivered back to the AWS server.

CREATING A TEXT CLASSIFIER MODEL

Before any of this is possible, a model must first be trained. Fortunately, a dataset containing manually scored comments (n = 95,851) was publicly available for download (Kaggle, 2017). This dataset contained binary target variables, flagging each text document as containing toxic, severely toxic, obscene, insulting, threatening, and hateful language. This data was uploaded to the CAS server. The TEXTMINE procedure was first used to both parse and filter the data. This process breaks up each

comment into individual terms, apply a stemming algorithm, and also filter out commonly used words. The next step is to use the BOOLRULE procedure to generate the rules used to classify the text comments. The code used to complete this process can be viewed below:

```
proc textmine data=mycaslib.train;
    var comment_text;
    doc_id id;
    parse outparent=mycaslib.parent outterms=mycaslib.terms
          outconfig=mycaslib.config;
run;

proc boolrule data=mycaslib.parent docinfo=mycaslib.train
          docid=_document_ terminfo=mycaslib.terms termid=_termnum_;
    docinfo id=id targets=(toxic severe_toxic obscene threat insult
          identity_hate) events=('1' '1' '1' '1' '1' '1');
    terminfo id=key label=term;
    output rules=mycaslib.rules ruleterms=mycaslib.ruleterms
          candidateterms=mycaslib.candidateterms;
run;
```

For the TEXTMINE procedure, it is important to save a number of output datasets using the parse statement. Information on the document collection terms, parent terms, and configuration options are required for both later modeling and scoring. On a similar note, in order to later score future text documents, tables created by the BOOLRULE procedure for the selected candidate rule terms, rules generated, and terms in each rule should be saved to a permanent location on the CAS server.

SAS Viya allows for analytical procedures to run across other programming languages. In this instance, a program written in Python will take the prior models developed in SAS Studio and process new data. It is important to be aware that each programming language may have differing rules that must be followed in order for it to execute. Python has case sensitivity and variable naming requirements which must be met. SAS code was produced in order to rename the variables in the tables to become ones required by the SAS SWAT package for Python:

```
data mycaslib.parent;
    set mycaslib.parent;
    rename _termnum_=_Termnum_ _document_=_Document_ _count_=_Count_;
run;

data mycaslib.terms;
    set mycaslib.terms;
    rename term=_Term_ role=_Role_ attribute=_Attribute_ freq=_Frequency_
          numdocs=_NumDocs_ _keep=_Keep_ key=_Termnum_ parent=_Parent_
          parent_id=_ParentId_ _ispar=_IsPar_ weight=_Weight_;
run;

data mycaslib.config;
    set mycaslib.config;
    rename language=_Language_ stemming=_Stemming_ tagging=_Tagging_
          ng=_NounGroup_ entities=_Entities_ cellwgt=_CellWeight_
          multiterm=_Multiterm_;
run;
```

```
data mycaslib.ruleterms;
    set mycaslib.ruleterms;
    rename target=_Target_ target_id=_TargetId_ target_var=_TargetVar_
          target_val=_TargetVal_
          ruleid=_RuleId_ rule=_Rule_ _termnum=_TermId_
          direction=_Direction_ weight=_Weight_;
run;
```

WORKER PROGRAM INTERACTION

After a model had been developed, it was then used to score incoming information and pass the results back to the server. As stated earlier, Python worker programs are already in place, continuously harvesting social media data and storing it. For this example, a Python worker had been developed to take the model developed and classify new, incoming data. In order to do this as close to real-time as possible, an intricate file delivery process was created. This solution allows each task to be specialized and separate from one another. The benefit of this technique provided the system added modularity resulting in resilience against system wide disruption when creating new features or fixing bugs.

An illustration of these interactions between worker programs can be seen in Figure 2. The syntax for the Python SAS SWAT worker can be found in APPENDIX 1 and numerical references to it are provided in this paper.

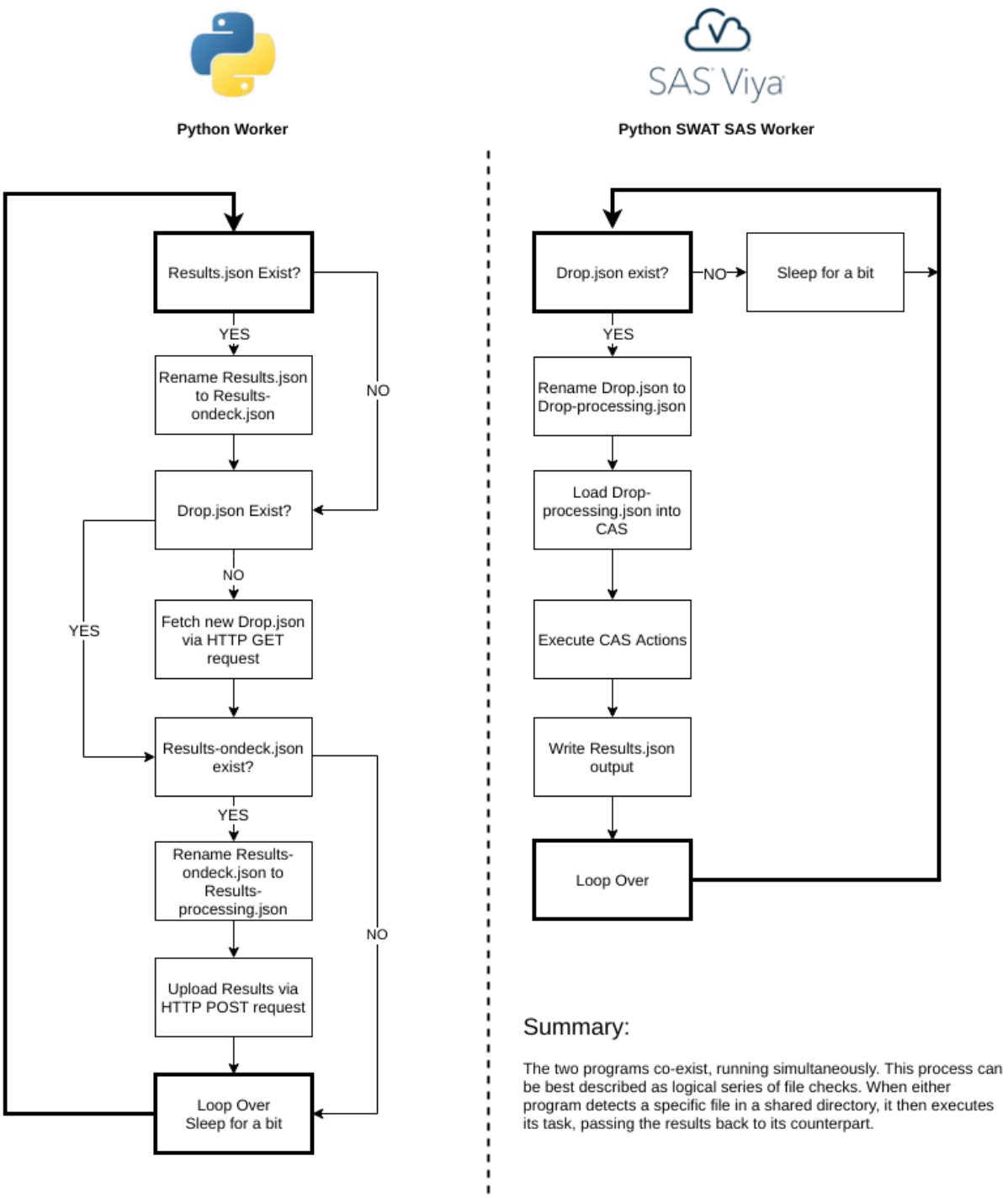

Figure 2. Flowchart of Worker Program Logic

STEPS 1 - 3: FILE MANAGEMENT

A series of file management steps exist at the beginning of the Python worker program. The file structure of the data being passed between the environments is in a JSON format. In order to work with this file type, the JSON and OS packages are imported and used. The SWAT module for data message handlers is also used to help pass data from the JSON file to the CAS server ❶.

The logic needed begins with the first step, detecting the presence of new data that needs to be scored. In this example, the Python worker that is collecting social media content places raw, unscored text documents into a .json file at the specified directory. When this file exists, the program will then execute the second step, which is to rename the file ❷. If the file does not exist, it will stop checking for a few seconds and try again ❸.

The reasoning behind this logic was to allow the first worker to then start preparing a completely new file to the directory. By doing so, the interactions between the two workers can improve efficiency and work together seamlessly getting to as close as real-time as possible.

After this logic has finished, the data was loaded into the CAS server for processing. The section of code that is responsible for this part of the program is as follows:

```
if not os.path.exists(FILE_PATH+FILE_NAME+FILE_TYPE):
          print("Can't find file...waiting (5 seconds)")
          time.sleep(5) ❸
          continue
    processed_files += 1
    print("Processing new file (#"+str(processed_files)+")")
    os.rename(FILE_PATH+FILE_NAME+FILE_TYPE,
    FILE_PATH+FILE_NAME+'-'+'processing.json') ❷
```

STEP 4: PROCESSING THE FILE

Now that the file exists on the CAS server, it can be scored using a combination of two different action sets available in Python ❹. The Text Mining action set gives access to the tmScore action, while the Boolean Rule action set provides the brScore action. The first set of code that must be run is the tmScore action. This procedure uses information in tables that relate to how the original training data was parsed and filtered to create its relevant terms. In order for this action to run, datasets for the terms, configuration, and parent information must exist on the CAS server. Other relevant parameters that must be provided are the document id and text field names:

```
s.textMining.tmScore(
    docId='id',
    documents={'caslib':'casuser', 'name':'to_score'},
    parseConfig={'caslib':'casuser', 'name':'config'},
    terms={'caslib':'casuser', 'name':'terms'},
    parent={'caslib':'casuser','name':'parent_score'},
    text='comment_text'
)
```

The second part of the scoring process includes the brScore action. Originally, the BOOLRULE procedure was used to identify the terms most likely associated with the target variable. This time around, Python code is using the tokenized output and applying the Boolean rules to it. Information indicating the document id variable, term id variable, rule terms, data to score, and which CAS table to store the results must be provided:

```
s.boolRule.brScore(
    casOut={'caslib': 'casuser', 'name': 'scored'},
    docId='_Document_',
    ruleTerms={'caslib': 'casuser', 'name': 'ruleterms'},
    table={'caslib': 'casuser', 'name': 'parent_score'},
    termId='_Termnum_'
)
```

This information is then transposed and merged back into the original text comments using the document ID ❺. An example of what this data might look like can be found in Table 1 below.

Display 1. Example Output of Scored Documents

doc_id	toxic	severe_toxic	obscene	threat	insult	identity_hate
22256635.0	0	0	0	1	0	0
138560519.0	0	0	0	0	0	0
169740962.0	0	0	0	0	0	0
284253328.0	0	0	0	0	0	0
345843351.0	0	0	0	1	0	1
414397990.0	0	0	0	0	0	0
475961218.0	0	0	0	0	0	0
607698611.0	0	0	0	0	0	0
686065030.0	0	0	0	0	0	0
765142199.0	0	0	0	0	0	0
850753790.0	0	0	0	0	0	0
950706683.0	0	0	0	0	0	0
1048994919.0	0	0	0	0	0	0
1102537245.0	0	0	0	1	0	1

Table 1. Example Output of Scored Documents

STEP 5: EXPORT SCORED DATA

Now that the documents have been scored, the final step in this process is to export the scored documents back to a JSON file. The JSON package is once again used, but this time to convert from a Python object back to a properly formatted JSON string. This is done using the dumps function and the resulting file is then delivered back to the other worker program to pick up ❻:

```
with open(FILE_PATH+OUTPUT_FILE+FILE_TYPE,'w') as out:
        out.write(json.dumps(final_out))❻
```

STEP 6: CREATING AN INFINITE LOOP

The whole purpose of these worker programs is to run continuously fetching, processing, and delivering data to one another. While most SAS users will often run their programs on a set schedule or in a batch process, the solution proposed in this paper takes it a step further. All of the code that performs the file check, rename, processing, and exporting of the data is wrapped within a logical loop that is designed to never resolve ❼. By doing so, the worker program will run nonstop, processing the data as close to real-time as possible.

CONCLUSION

Overall, the creation and deployment of a text classification algorithm was a simple task using SAS Viya and the CAS server. As social media comments are being collected, they are immediately scored and the results are passed back to the original environment. Doing so allows for the categorization and tracking of toxic behavior. In today's environment, this can hopefully lead to a better understanding of online interaction and help foster healthy discussions to keep people engaged.

SAS Viya offers a wide variety of powerful analytical solutions to overcome everyday business challenges. The example provided in this paper helps demonstrate how data mining and machine learning worker programs can be developed and deployed, producing real-time analytic results. The SAS Viya platform helps break down barriers that exist between developers and allow greater access to analytics.

REFERENCES

Kaggle. 2017. “Toxic Comment Classification Challenge.” Accessed December 27, 2017. https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge.

RECOMMENDED READING

- SAS® Viya™: The Python Perspective
- SAS® Visual Data Mining and Machine Learning 8.2: Programming Guide

CONTACT INFORMATION

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

Scott Koval
Pinnacle Solutions, Inc.
(317) 423-9143
scott.koval@thepinnaclesolutions.com
www.thepinnaclesolutions.com

Mia Lyst
Pinnacle Solutions, Inc.
(317) 423-9143
mia.lyst@thepinnaclesolutions.com
www.thepinnaclesolutions.com

Jon Klopfer
Pinnacle Solutions, Inc.
(317) 423-9143
jon.klopfer@thepinnaclesolutions.com
www.thepinnaclesolutions.com

APPENDIX 1

```
import os, time, json, swat ❶
from pprint import pprint
from io import StringIO
import swat.cas.datamsghandlers as dmh

FILE_PATH = 'psi-viya-worker/mydata'
FILE_NAME = '/sas_work'
OUTPUT_FILE = '/sas_results'
FILE_TYPE = '.json'

TOXIC_TYPES = [
          'nothing',
          'toxic',
          'severe_toxic',
          'obscene',
          'threat',
          'insult',
          'identity_hate',
]

s = swat.CAS('host', 'port', 'user','password')

s.loadactionset(actionset="textmining") ❹
s.loadactionset(actionset="boolRule")

#Loading Tables
terms = s.loadtable('mydata/parsed_terms.sas7bdat', caslib='casuser')
config = s.loadtable('mydata/parseconfig.sas7bdat', caslib='casuser')
stoplist = s.loadtable('mydata/stoplist.sas7bdat', caslib='casuser')
parent = s.loadtable('mydata/term_by_doc.sas7bdat', caslib='casuser')
candidateterms = s.loadtable('mydata/tox_candidate_term.sas7bdat',
caslib='casuser')
rules = s.loadtable('mydata/tox_rules.sas7bdat', caslib='casuser')
ruleterms = s.loadtable('mydata/tox_rules_term.sas7bdat', caslib='casuser')

processed_files = 0
while True: ❼
    if not os.path.exists(FILE_PATH+FILE_NAME+FILE_TYPE):
          print("Can't find file...waiting (5 seconds)")
          time.sleep(5) ❸
          continue
    processed_files += 1
    print("Processing new file (#"+str(processed_files)+")")
    os.rename(FILE_PATH+FILE_NAME+FILE_TYPE, FILE_PATH+FILE_NAME+'-
'+'processing.json') ❷

    with open(FILE_PATH+FILE_NAME+'-processing.json','r') as infile:
          incoming = json.loads(infile.readline())

    raw_text = 'id,comment_text\n'
    for d in incoming['data']:
          raw_text+=','.join([str(d['id']), '"' +
d['text'].replace('"','').replace("\n"," ").replace("\r"," ").strip() +
'"']) + '\n'
```

```
    new_text = StringIO(raw_text)
    handler = dmh.CSV(new_text, skipinitialspace=True)
    s.addtable(table='tox_test', replace=True, **handler.args.addtable)

    s.textMining.tmScore(
          docId='id',
          documents={'caslib':'casuser', 'name':'tox_test'},
          parseConfig={'caslib':'casuser', 'name':'config'},
          terms={'caslib':'casuser', 'name':'terms'},
          parent={'caslib':'casuser','name':'sParent','replace':True},
          text='comment_text'
    )

    s.boolRule.brScore(
          casOut={'caslib': 'casuser', 'name':
'tox_new_score','replace':True},
          docId='_Document_',
          ruleTerms={'caslib': 'casuser', 'name': 'ruleTerms'},
          table={'caslib': 'casuser', 'name': 'sParent'},
          termId='_Termnum_'
    )

    output = s.fetch('tox_new_score', to=1000)

    results = json.loads(output['Fetch'].to_json())

    ret = {}

    #Loop over result items ❺
    for row, doc_id in results['_DocId_'].items():
          if not doc_id in ret:
                      ret[doc_id] = {}
                      for i in range(1,7):
                            ret[doc_id][float(i)] = 0
          value = results['_Target_'][row]
          if value:
                ret[doc_id][value] = 1

    final_out = []

    for k,v in ret.items():
          any_flag = 0
          for kk, vv in v.items():
          if vv == 1:
                any_flag = 1
                break
          final_out.append({
                'id': int(k),
                'tox_flag': any_flag,
                'text_topic': str(v),
          })

    with open(FILE_PATH+OUTPUT_FILE+FILE_TYPE,'w') as out:
          out.write(json.dumps(final_out)) ❻
```