Demo Steps: Using the Python API to Explore Text Documents

The data set **CFPB_COMPLAINTS** was obtained from the Consumer Financial Protection Bureau (https://www.consumerfinance.gov/data-research/consumer-complaints/). It contains complaints filed by consumers and a binary indicator indicating whether the consumer disputed the company's response to the claim. The goal of this analysis is first to analyze the raw unstructured text and then build a model to predict when a consumer will dispute the company's response from the complaint.

Name	Model Role	Measurement Level	Description
DISPUTE	Target	Binary	1 = consumer disputed company response 0 = did not dispute
COMPLAINT	Input	Text	consumer submitted complaint

1. From Jupyter Lab, select **File Browser** > **Home** > **Courses** > **EVMLOPRC** > **Notebooks** and select the **Python_Text_Mining_Demo.ipynb** notebook.

2. Load the os, sys, SWAT, numpy, pandas, and matplotlib packages.
```
import os
import sys
import swat
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
swat.options.cas.print_messages = True
```
3. Connect to CAS and create a connection object to use in subsequent code.
```
conn = swat.CAS(os.environ.get("CASHOST"), os.environ.get("CASPORT"), None, os.environ.get("SAS_VIYA_TOKEN"))
```
4. Load the **cfpb_complaints.sas7bdat** data set onto the CAS server with the upload action and name it **cfpb**. The upload action does not create a table reference when moved to the CAS server. Create a table reference in order to use SWAT functionality on the CAS table.
```
uptab =
   conn.upload(os.environ.get("HOME")+"/Courses/EVMLOPRC/DATA/cfpb_complaints.sas7bdat",
   casout = dict(name="cfpb", replace=True))

indata = 'cfpb'
castbl = conn.CASTable(name = indata)
```
5. Find the dimension of the data and print the first few observations.
```
display(castbl.shape)
pd.options.display.max_colwidth=10000
castbl.head(2)
```
6. Use a DATA step to give the data a minor clean. Remove all non-letters from the text and reduce letters to lowercase. Also, add a sequence variable to represent the document identification.
```
conn.loadActionSet('dataStep')
actions = conn.builtins.help(actionSet='dataStep')

conn.dataStep.runCode(code=
    '''
    data cfpb;
        set cfpb;
        complaint = lowcase(compress(complaint,
           'ABCDEFGHIJKLMNOPQRSTUVWXYZ.!?1234567890 ', 'ki'));
        complaint = tranwrd(complaint, ' xxxx', '');
        docid + 1;
    run;
    '''
)

conn.table.fetch(table='cfpb', to=5)
```
Selected functions:

Name	Description
lowcase	converts all letters in a string to lowercase.
compress	returns a character string with specified characters that were removed from the original string. The first argument is the variable or string from which specified characters are removed. The second argument lists the characters that are used by the third argument, the modifier. The K modifier keeps the characters in the list instead of removing them, the I modifier ignores the case of characters to be kept or removed, and the S argument adds space characters to the list of characters.

The **docid** variable is used in a sum statement. The variable is automatically initialized to zero and its value is retained from one iteration of the DATA step to the next. On each iteration, the new variable is incremented by the sequence value.

7. Use a DATA step to find all documents with the term *debt* and subset those documents into a new CAS table. Print the number of records and the first few instances of the found documents.
```
debt = conn.dataStep.runCode(code=
    '''
    data debt (drop=newvar);
```

```
        set cfpb;
        newvar = find(complaint,'debt','i');
        if newvar > 0;
    run;
    '''
)

display(conn.table.recordCount(table='debt'))
conn.table.fetch(table='debt', to=5)
```
Selected function:

Name	Description
find	searches for a specific substring of characters within a character string. The first argument is the string or variable. The second is the substring to find. The third argument, I, is a modifier that instructs the function to ignore case.

The find function returns either the location of the substring or a zero if it is not in the variable. The IF statement then keeps the observation if the substring is present.

8. Use a DATA step to find all the documents with the term *lawyer*, and subset those documents into a new CAS table. Print the number of found documents and the first few instances.

```
lawyer = conn.dataStep.runCode(code=
    '''
    data lawyer (drop=newvar);
        set cfpb;
        newvar = find(complaint,'lawyer','i');
        if newvar>0;
    run;
    '''
)

display(conn.table.recordCount(table='lawyer'))
conn.table.fetch(table='lawyer', to=5)
```

9. Use the tpParse action from the textParse action set to parse the terms in the documents. Stem terms and avoid noun groups, entities, and tagging. Save the parse configuration and offset tables and view them with the fetch action.

```
conn.loadActionSet('textParse')
actions = conn.builtins.help(actionSet='textParse')

conn.textParse.tpParse(
    table = indata,
    docid = 'docid',
    text = 'complaint',
    stemming = True,
    nounGroups = False,
    entities = 'none',
    tagging = False,
    parseConfig = dict(name='config', replace=True),
    offset = dict(name='offset', replace=True)
)
```
Selected arguments:

Argument	Description
docid	specifies the character or numeric variable in the table that contains the ID of each document.
text	specifies, in the documents, the character variable that contains the text to be processed.
stemming	specifies whether stemming is to occur during parsing. When set to *TRUE*, terms are evaluated to see whether they belong to common parent form and information is added to the position table.
nounGroups	specifies that noun group extraction is to occur during parsing. When set to *TRUE*, noun groups become additional rows on the position table. This is also reflected in the terms and parent tables.
entities	specifies whether to extract entities in parsing. If the value is set to *STD*, the standard entities are written to the output. An *entity* is any of several types of information that SAS can distinguish from general text. SAS identifies the following standard entities: ◦ ADDRESS (postal address or number and street name) ◦ COMPANY (company name) ◦ CURRENCY (currency or currency expression) ◦ DATE (date, day, month, or year) ◦ INTERNET (email address or URL) ◦ LOCATION (city, country, state, geographical place or region, political place or region) ◦ MEASURE (measurement or measurement expression) ◦ ORGANIZATION (government, legal, or service agency) ◦ PERCENT (percentage or percentage expression) ◦ PERSON (person's name) ◦ PHONE (phone number) ◦ PROP_MISC (proper noun with an ambiguous classification) ◦ SSN (Social Security number) ◦ TIME (time or time expression) ◦ TIME_PERIOD (measure of time expressions) ◦ TITLE (person's title or position) ◦ VEHICLE (motor vehicle including color, year, make, and model)
tagging	specifies whether part-of-speech tagging is used in parsing.

parseConfig	specifies the name of the config CAS table to contain parsing configuration information.
offset	specifies the name of the output CAS table to contain the position information about the occurrences of child terms in the document collection. Child terms are associations to parent terms. For example, in restaurant reviews, the term *price* could be a parent term and a related child term could be *fast-food*.

```
display(conn.table.fetch(table='config'))
conn.table.fetch(table='offset', to=5)
```

The default language is English. It can be changed using the language argument in the tpParse action. The config table provides information about how the parsing was executed. The offset table provides position and attribute information for each term in the documents table. This is used to accumulate text in other actions.

10. Upload a list of stop words from the stop_words.csv file and print the terms along with the total number of terms.

```
stoplist = conn.read_csv(
   os.environ.get("HOME")+"/Courses/EVMLOPRC/DATA/stop_words.csv",
   casout = dict(name="stoplist", replace=True))

print('Upload a list of '+ str(stoplist.shape[0])
       +' stop words.')
stoplist.head(stoplist.shape[0])
```

Note: This stop word list is based on the SMART (System for the Mechanical Analysis and Retrieval of Text) Information Retrieval System, which is an information retrieval system that was developed at Cornell University in the 1960s.

11. Using the tpAccumulate action from the textParse action set, accumulate the parsing results. Use the stop list previously loaded, stem terms, avoid tagging, and keep all terms without a minimum number of instances. Finally, save the parent, child, and terms table and view their contents.

```
conn.textParse.tpAccumulate(
    stopList = 'stoplist',
    stemming = True,
    tagging = False,
    reduce = 1,
    offset = 'offset',
    showDroppedTerms = False,
    parent = dict(name='parent', replace=True),
    child = dict(name='child', replace=True),
    terms = dict(name='terms', replace=True)
)
```

Selected arguments:

Argument	Description
stoplist	specifies the input CAS table that contains the terms to exclude from the analysis. If it is specified, the table must have the term variable. A role variable is optional.
stemming	specifies whether stemming is to occur during parsing. When set to *TRUE*, terms are evaluated to see whether they belong to a common parent form. This information is added to the position table.
tagging	specifies whether part-of-speech tagging is used during parsing.
reduce	specifies the minimum number of documents in which a term should occur if it is retained.
offset	specifies the name of the input CAS table that contains the position information about the occurrence of child terms in the document collection.
showDroppedTerms	specifies whether to include terms that have a keep status of *N* in the OUTTERMS output table.
parent	specifies the name of the output CAS table that contains a compressed representation of the sparse term-by-document matrix.
child	specifies the name of the output CAS table that contains a compressed representation of the sparse term-by-document matrix with raw counts.
terms	specifies the name of the output CAS table that contains the summary information about the terms in the document collection.

```
conn.table.fetch(table='terms', to=5
```

Only the terms output CAS table contains the actual term (**_Term_**), but the child and parent tables contain document location (**_Document_**) and count information for the term identification number (**_Termnum_**). The **_Keep_** variable indicates whether the term is kept for analysis or is removed due to the stop words list. The attributes of the term give an indication of the characters that compose that term:

- Alpha, if characters are all letters
- Num, if term characters include a number
- Punct, if the term is a punctuation character
- Mixed, if term characters include a mix of letters, punctuation, and white space
- Entity, if the term is an entity

12. Create a table object reference to the terms table that was created above, and use the SWAT function unique to find the total number of unique terms. Then use a DATA step to create a table of the unique terms.

```
terms_tbl = conn.CASTable(name='terms')
print('The total number of terms = ' + str(terms_tbl.shape[0]))
print('The number of unique terms = ' +
       str(terms_tbl._Term_.unique().shape[0]))

conn.dataStep.runCode(code=
    '''
    data terms_unique;
        set terms;
        by _Term_;
        if last._Term_;
    run;
    '''
)
```

The terms table contains duplicate terms as a result of the natural language processing that is executed by SAS. Terms can be identified as multiple parts of speech. For example, the term *fair* can be either a noun or an adjective depending on the sentence: "People go to the county fair in October." versus "The team had a fair referee tonight." SAS identifies the following parts of speech:

- Abbr (abbreviation)
- Adj (adjective)
- Adv (adverb)
- Aux (auxiliary or modal)
- Conj (conjunction)
- Det (determiner)
- Interj (interjection)
- Noun (noun)
- Num (number or numeric expression)
- Part (infinitive marker, negative participle, or possessive marker)
- Pref (prefix)
- Prep (preposition)
- Pron (pronoun)
- Prop (proper noun)
- Punct (punctuation)
- Verb (verb)
- VerbAdj (verb adjective)

A term used in multiple parts of speech can have multiple entries in the terms table. The same term is located consecutively. This makes it easy to use a DATA step to filter out only the unique terms.

The output indicates that of the 26698 terms in the table, only 23915 are unique.

13. Use SQL to find the top 250 most used terms in the data set and save them as a new data table on the server named **top_terms**.

```
conn.loadActionSet('fedSql')
actions = conn.builtins.help(actionSet='fedSql')


conn.fedSql.execDirect(query =
    '''
    CREATE TABLE top_terms AS
    SELECT _Term_, _Frequency_
    FROM terms_unique
    ORDER BY _Frequency_ DESC
    LIMIT 250;
    '''
)
```

14. Download the **top_terms** data table to the client and create a bar chart for the top 10.

```
top_terms = conn.CASTable(name='top_terms')
top_terms = top_terms.to_frame()

top_terms.columns = ['term','freq']
display(top_terms.head())
top_terms.shape

TermPlot = top_terms.head(10).plot(x='term', y='freq',
   kind='bar', figsize=(8,8), fontsize=15, color='blue')
TermPlot.set_xlabel('')
TermPlot.set_ylabel('Frequency', fontsize=15)
TermPlot.legend_.remove()
plt.show()
```

15. Use the tmMine action from the textmining action set to discover topics within the documents. Find ten topics from the complaints data set and save the topics as a CAS table.

```
conn.loadActionSet('textmining')
actions = conn.builtins.help(actionSet='textmining')

conn.textMining.tmMine(
    documents = indata,
    docid = 'docid',
    text = 'complaint',
    nounGroups = False,
    tagging = False,
    stemming = True,
    stopList = 'stoplist',
    reduce = 1,
    k = 10,
    numLabels = 10,
    topicDecision = True,

    # Save same tables from the tpParse Action
    parseConfig = dict(name='config', replace=True),
    parent = dict(name='parent', replace=True),
    child = dict(name='child', replace=True),
    offset = dict(name='offset', replace=True),
    terms = dict(name='terms', replace=True),

    # Save tables from the tmMine action
    termTopics = dict(name='term_topics', replace=True),
    wordPro = dict(name='wordpro', replace=True),
    docpro = dict(name='docpro', replace=True),
    topics = dict(name='topics', replace=True),
    u = dict(name='svdu', replace=True),
    s = dict(name='singular_vals', replace=True)
)
```

The tmMine action from the textmining action set contains all the same functionality as the textParse action set. The textmining action set provides additional functionality to create topics.

Selected arguments:

Argument	Description
documents	names the input CAS table of documents to be parsed. You must include a text variable that is specified with textVar and a document ID variable that is specified with docIdVar.
docid	specifies the character or numeric variable in the documents table that contains the ID of each document.
text	specifies the character variable in the documents table that contains the text to be processed.
nounGroups	specifies that noun group extraction is to occur during parsing. When set to *TRUE*, noun groups become additional rows on the position table. This is reflected in the terms and parent tables.
tagging	specifies whether part-of-speech tagging is used during parsing.
stemming	specifies whether stemming is to occur during parsing. When set to *TRUE*, terms are evaluated to see whether they belong to a common parent form. The information is added to the position table.
stopList	specifies the input CAS table that contains the terms to exclude from the analysis. If it is specified, the table must have the term (varchar) variable. A role (varchar) variable is optional.
reduce	specifies the minimum number of documents in which a term should occur if it is to be retained.
k	specifies the number of dimensions to be extracted (also the number of derived topics). If the input data is too small for the requested number of dimensions, this value is adjusted to complete the calculation.
numLabels	specifies the number of terms to use in the descriptive label for each topic.
topicDecision	specifies to include topic membership decisions and document cutoffs in the output tables.
parseConfig	specifies the name of the config CAS table that contains parsing configuration information.
parent	specifies the name of the output CAS table that contains a compressed representation of the sparse term-by-document matrix.
child	specifies the name of the output CAS table that contains a compressed representation of the sparse term-by-document matrix with raw counts.
offset	specifies the name of the output CAS table that contains the position information about the occurrences of child terms in the document collection.
terms	specifies the output CAS table that contains the summary information about the terms in the document collection.
termTopics	specifies the name of the output CAS table that contains the term-by-topic sparse matrix information.
wordpro	specifies the table that contains the projections of the terms. If *k* dimensions of the SVD are found and the input data set contains *n* terms, this table has *n* rows and *k*+1 columns.
docpro	specifies the name of the table that contains the SVD projections of the documents.
topics	specifies the output CAS table that contains the topics that are discovered.
u	specifies the **U** matrix, which contains the left singular vectors. The matrix **U** is the number of terms by *k*+1.
s	specifies the **S** matrix, which is a diagonal matrix that is written to output in compressed form, with two variables and *k* rows. The variable **_ID_** indicates the row and column of the entry and the variable **S** contains the singular values.

```
conn.table.fetch(table='topics',to=5)
```

The plus symbol that is connected to terms indicates that it was stemmed. The number of topics can be changed to accommodate a broader number of areas in the reviews by changing the *k* argument in the tmMine action. For simplicity, only five were chosen for the demonstration.

The topics in this example did a poor job of classifying documents because the **_NumDocs_** variable is zero for each topic. For each created topic, the algorithm computes a topic weight for every term in the corpus (**_TermCutOff_**). This measures how strongly the term represents or is associated with the given topic. Terms that have topic weights above the term cutoff are associated with the topic. If the topic weight exceeds the document cutoff (**_DocCutOff_**), then the document is classified as having the topic.

16. Use SQL to find the term number for the term *great*. Then use the tmFindSimilar action from the textUtil action set to find terms that are similar to *loan*.

```
conn.loadActionSet('textUtil')
actions = conn.builtins.help(actionSet='textUtil')

termnum = conn.fedSql.execDirect(query =
    '''
    SELECT _Termnum_
    FROM terms_unique
    WHERE _Term_ = 'loan';
    '''
)['Result Set']
```

```
termnum = termnum['_Termnum_'][0]
termnum

conn.textUtil.tmFindSimilar(
    table = 'wordpro',
    termnum = termnum,
    num_svd = '5',
    prefix = 'col',
    casOut = dict(name='similar', replace=True)
)

conn.table.fetch(table='similar', to=5)
```

The tmFindSimilar action calculates the similarity scores of a specified term with every other term in the collection. The action applies a cosine similarity calculation to the WORDPRO table of the tmMine action to produce these similarity scores. Because the term projections are based on the co-occurrence patterns in the corpus, higher similarity scores are assigned to terms that have more closely related occurrence patterns with the specified term.

Selected arguments:

Argument	Description
table	specifies the input table that contains the SVD projections of the terms.
termnum	specifies a term ID that is used so that the similarity can be scored against terms in the input table.
num_svd	specifies how many dimensions of the SVD projections are used in the similarity computation. The first *k* columns of SVD projections are used.
prefix	specifies the prefix of variables that contain the SVD projections.
casOut	specifies the output CAS table that contains the similarity scores of the terms.

17. Use a DATA step to merge the terms, term numbers, and similarity scores. Then use SQL to print the most similar terms.

```
conn.dataStep.runCode(code=
    '''
    data similar_terms (keep = Term _TermNum_ SimNum);
        merge terms_unique(IN=in1 rename=(_Term_=Term))
           similar(IN=in2 rename=(_Similar_=SimNum));
        by _TermNum_;
        if (In1=1 and In2=1);
    run;
    '''
)
```

Because the similar scores contain only the term number identifications, you need to merge the tables by the **_Termnum_** variable to find the actual string.

```
print("Most Similar Terms")
simTerms = conn.fedSql.execDirect(query =
    '''
    SELECT *
    FROM similar_terms
    ORDER BY SimNum DESC;
    '''
)['Result Set']

simTerms.head()
```