Paper SAS4442-2020

How to Build a Text Analytics Model in SAS® Viya® with Python

Nate Gilmore, Vinay Ashokkumar, and Russell Albright, SAS Institute Inc.

ABSTRACT

Python is widely noted as one of the most important languages influencing the development of machine learning and artificial intelligence. SAS® has made seamless integration with Python one of its recent focal points. With the introduction of the SAS® Scripting Wrapper for Analytics Transfer (SWAT) package, Python users can now easily take advantage of the power of SAS® Viya®. This paper is designed for Python users who want to learn more about getting started with SAS® Cloud Analytic Services (CAS) actions for text analytics. It walks them through the process of building a text analytics model from end to end by using a Jupyter Notebook as the Python client to connect to SAS Viya. Areas that are covered include loading data into CAS, manipulating CAS tables by using Python libraries, text parsing, converting unstructured text into input variables used in a predictive model, and scoring models. The ease of use of SWAT to interact with SAS Viya using Python is showcased throughout the text analytics model building process.

INTRODUCTION

One of the ways that SAS provides access to its high-quality text analytics services is through CAS actions that can be invoked directly from SAS, Python, Lua, or R. These actions cover a wide array of functionality including text mining, text categorization, concept identification, and sentiment analysis.

This paper highlights the construction of an end-to-end text analytics model that leverages CAS actions called from Python. It demonstrates how the unstructured text of Amazon reviews can be converted into structured input variables and used in a Support Vector Machine (SVM) model to predict whether users will find an Amazon review helpful. The client-side platform used is a Jupyter notebook, a very popular interface for Python users. The version of Python used for this project was Python 3.4.1.

PYTHON AND CAS

In recent history, many users have embraced Python as a first-choice programming language for the development of software across all domains. The community has embraced Python's open source nature and ease of use through many functional libraries to aid in design and performance. As such, the Python user space continues to grow unfailingly.

SAS Viya has opened its arms to Python users by allowing integration of open source to its platform and its use of features. SAS has introduced SWAT (Scripting Wrapper for Analytics Transfer), a Python library that enables users, even those with no SAS background, the ability to continue coding in Python, while leveraging the performance and resources of CAS and SAS Viya in their applications. Users are able to create, manipulate, and print data with CAS actions through the Python interface. Added performance improvements are due to the data being processed while on the cloud as a part of the SAS Viya architecture and the data being loaded and worked on in memory. Other popular libraries, like pandas and NumPy, are supported to allow increased compatibility with the open source tools.

Figure 1 below shows the Python libraries that are imported and used in this project. Under that, the basic syntax for connecting to CAS through SWAT is shown, including using the user's credentials. The final lines of code define and load all the action sets used in this project as a list. The relevant CAS actions under each respective action set will be detailed in the upcoming sections.

```
#Importing required packages and modules
import swat.datamsghandlers as dmh
from io import StringIO #Needed for creating an in-line synonym list
import pandas as pd #Allows manipulation of data through Data Frames
import numpy as np #Allows creation and manipulation of arrays and matrices
import matplotlib.pyplot as plt #Supports 2D plotting, used in this project for generating ROC Curve

# Connect to CAS Server
s = swat.CAS(host, port, username, password)

# Load all CAS action sets required to complete the model building process
action_sets = ['sampling', 'fedSQL', 'textParse', 'sampling', 'textMining', 'textUtil', 'svm', 'astore', 'percentile']
[s.builtins.loadactionset(i) for i in action_sets]
```

Figure 1. Importing Libraries, Connecting to CAS, and Loading Action Sets

USE CASE – PREDICTING REVIEW HELPFULNESS

The data set analyzed in this paper includes over 67,000 Amazon reviews of fine food products. The model built in this project will predict whether a review will be rated helpful or not by Amazon users. For the purposes of this paper, a helpful review is defined as one that at least 80% of voters found helpful (with a minimum of 5 users having voted on its helpfulness). The explanatory variables considered include the star rating of the review, the length of the review, and the text of the reviews. The text analytics portion of the model building process focuses on converting the unstructured text of the review into document projections that will be used as input variables to the SVM predictive model along with the star rating and review length. The Analyzing Results section of this paper details how including document projections derived from the review text significantly improves the predictive model's accuracy compared to using only star rating and review length. Table 1 below describes the most relevant variables in the data set, which will be referenced in the code snippets of upcoming sections.

Variable	Description
ID	A unique identifier for each review
Text	Text of the product review
Score	Star rating for a review (From 1 to 5)
Review_Length	Number of words in a review
Helpful	Target Variable (1=Helpful, 0=Not Helpful)

Table 1. Description of Relevant Variables

LOADING DATA INTO CAS

The data preprocessing steps necessary to prepare the data for consumption by the model were performed in Python on the client side prior to loading data into CAS. Those steps included:

1. Dropping all observations that did not have at least five helpfulness ratings.
2. Creating a predictor variable containing the number of words in a review.

3. Creating a unique identifier variable for each review as required by the text actions.
4. Defining a target variable for review helpfulness as indicated in the previous section.

After preprocessing, the first step in preparing the data to be loaded into CAS is to define a casLib to the location where your data is stored. Figure 2 shows how to use the addCaslib action to define a caslib named "projectData " in the location where the input data is stored.

```
#Add caslib for location where project data is saved as a sashdat
s.table.addCaslib(
name="projectData", path="/your/data/path", datasource={"srctype":"path"}, activeOnAdd=False)
```

```
NOTE: Cloud Analytic Services added the caslib 'projectData'.
```

Figure 2. addCaslib Action Code

After setting up the caslib, the loadTable action makes it easy to load your data to CAS. Figure 3 shows how to load the preprocessed Amazon Fine Food Reviews data set, stored as a .sashdat, into CAS. The data is stored as a CAS table named "amazonFull".

```
#Use the loadTable action to load input table
s.table.loadTable(casout={"name":"amazonFull"},
                  caslib="projectData", importoptions={"fileType":"HDAT"},
                  path="amazon_preprocessed_full.sashdat")
```

Figure 3. Loading Amazon Reviews into CAS with the loadTable Action

SPLITTING DATA INTO TRAINING AND VALIDATION SETS

Now that the model's input data has been loaded into CAS, the next step is to split the data into training and validation sets. For this project, 70% of reviews were included in the training set while 30% were reserved for validation. Reviews were assigned to either the training or validation data set via simple random sample using the srs CAS action as shown in Figure 4.

```
: # Create a 70% Training/30% Validation simple random split:
  s.sampling.srs(
      table={"name":"amazonFull"},
      samppct = 70,
      partind = True,
      seed    = 1,
      output = {"casOut":{"name":"split"}, "copyVars":"ALL"}
  )
```

Figure 4. Using the SRS Action to Create 70% Training/30% Validation

The srs action assigns a partition index, *_PartInd_*, which takes a value of 1 for reviews assigned to the training set and 0 for reviews assigned to the validation set. The execDirect action from the fedSQL action set is used to create separate CAS tables for the training and validation sets as shown in Figure 5.

```
#Splitting training and validation sets into separate tables
s.fedSQL.execDirect('''
  CREATE TABLE "amazonTraining" AS SELECT * FROM SPLIT WHERE "_PartInd_" = 1
''')
s.fedSQL.execDirect('''
  CREATE TABLE "amazonValidation" AS SELECT * FROM SPLIT WHERE "_PartInd_" = 0
''')
```

Figure 5. Using the execDirect Action to Split Training and Validation into Separate CAS Tables

DATA EXPLORATION

The next step in the model building process is to explore your training data. Initial data exploration shows that 63.7% of the training data set is comprised of helpful reviews while the remaining 36.3% of reviews are unhelpful. Further exploration shows that 50% of the reviews were rated 5 stars, 24% are rated 1 star, and the remaining 26% are split rather evenly between 2, 3, and 4 stars. One noteworthy finding is that while 86.2% of 5-star reviews are considered helpful, only 27.5% of 1-star reviews are considered helpful. This exploration process gives you an idea that the star rating will likely be a very useful variable in determining the likelihood that a review is helpful. Users are more likely to consider reviews with higher star ratings as helpful than those that have lower star ratings. Figure 6 shows the code to perform a cross-tabulation using the pandas library along with the resulting stacked bar chart that demonstrates how the proportion of reviews that are considered helpful varies for each level of review rating.

```
#Converting CAS Table into CAS Table Object (Intermediate data structure required prior to converting to data frame)
training = s.CASTable("amazonTraining")

#Converting the training data to a data frame
df_training = training.to_frame()

#Performing a cross-tabulation of Star Rating and Review Helpfulness
tabulation = pd.crosstab(df_training.score, df_training.helpful)

#Add column Headers
tabulation.columns = ['Unhelpful Total', 'Helpful Total']

#Creating Stacked Bar Chart
tf = tabulation.plot.bar(stacked=True, title="Review Helpfulness by Review Rating")
tf.set_xlabel("User Rating (Stars)")
tf.set_ylabel("Number of Observations")
```

```
Text(0,0.5,'Number of Observations')
```

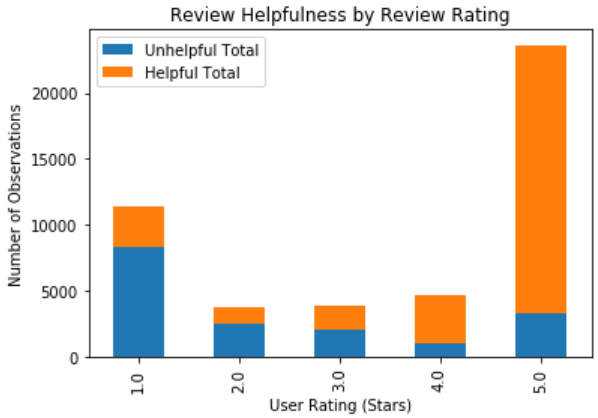

Figure 6. Relationship between Review Rating and Review Helpfulness

MODEL TRAINING

This section shows the various steps in building a predictive model that includes text. The code not only includes a standard predictive model, in this case we use an SVM, but also the actions needed to transform your unstructured data to a numeric representation.

Figure 7 shows the general overview of the code needed to create the models for scoring. There are two major components, one for text and the other for the predictive model. Both sections produce their own analytic store model. In the following subsections, these model training components are covered more specifically.

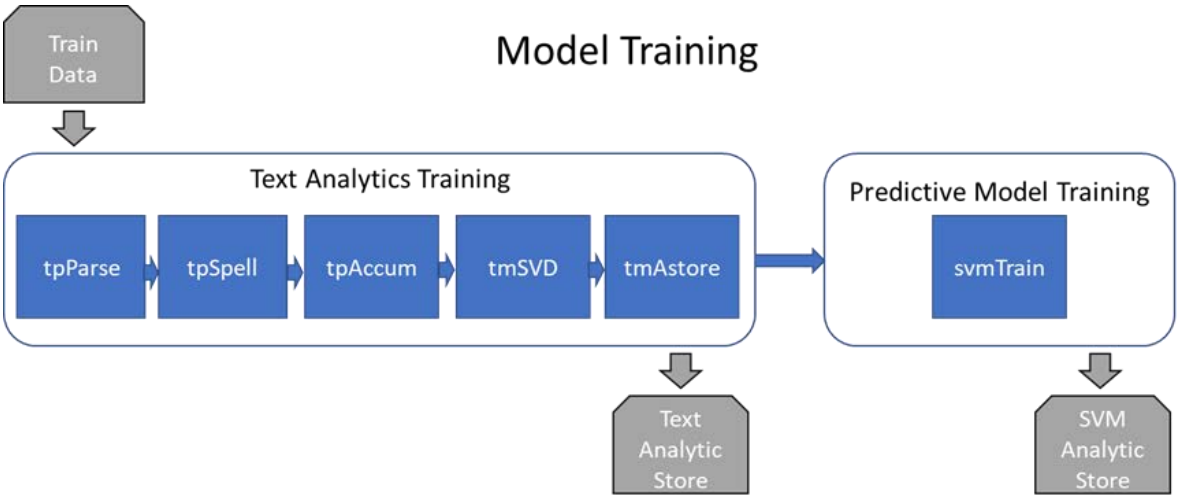

Figure 7. Model Training Overview

PARSING THE TEXT

The tpParse action parses each row of the input table and creates an output offset table that lists every term found in every document. It is the first step in transforming your text to a numerical representation. There are many options to control how the tokenization works and to enable you to use various natural language features such as the part-of-speech tags or the stemmed form of a term. In the code shown in Figure 8, you can see common settings used in the tpParse action. You should explore which settings work best for your particular input data and subsequent model.

```
#Call tpParse action to parse the text of each review
parsed_results=s.tpParse(table={"name":"amazonTraining"},
                  docid="id",
                  text="text",
                  entities="std",
                  noungroups=True,
                  stemming=True,
                  tagging=True,
                  outComplexTag=True,
                  predefinedMultiterm=True,
                  language="English",
                  offset={"name":"offset", "replace":True},
                  parseConfig={"name":"outConfig", "replace":True}
                  )
```

Figure 8. The tpParse Action Code

In addition to the output offset table, you should also request the parseConfig output. This table stores the settings that you use so that they can be reused at score time. Below, you will see that the parseConfig table is added to and then used to build a scoring model.

CORRECTING MISSPELLINGS

When your input text data is particularly noisy, such as informal chat messages or other unedited content, the tpSpell action can be useful for automatically correcting misspellings. This action takes the offset table from the tpParse action, analyzes it for spelling corrections that need to be made, and, on output, updates the offset table with these corrections. The tpSpell action finds candidate misspellings by looking across the entire collection for rare terms that are very similar in spelling to more common terms. The code for calling the tpSpell action is shown in Figure 9.

```
#Running tpSpell action with default settings
spell_results=s.tpSpell(table={"name":"offset"},
                  casOut={"name":"offset_spell", "replace":True}
                  )
```

Figure 9. The tpSpell Action Code

In Figure 10, you can see the fetch action that retrieves and displays a subset of the output table from the tpSpell action. This output table replaces the offset table of tpParse, correcting the parent values of misspelled terms. In the table shown in Figure 10, the misspelled term "allert" has been corrected to having a parent of "alert".

```
#Viewing a sample spelling correction for term 'allert' corrected to a parent of 'alert'
d = s.table.fetch(table={"name":"offset_spell","where":"_document_ = 22"},sortBy=[
        {"name":"_term_", "order":"ascending"}], to=6)
d
```

§ Fetch

Selected Rows from Table OFFSET_SPELL

	Term	_Role_	_ComplexTag_	_Attribute_	_Parent_	_Start_	_End_	_Sentence_	_Paragraph_	_Document_
0	a	DET	Det	1.0	a	180.0	180.0	2.0	0.0	22.0
1	after	ADV	Adv	1.0	after	260.0	264.0	2.0	0.0	22.0
2	all	ADV	Adv	1.0	all	328.0	330.0	3.0	0.0	22.0
3	allergy	N	N	1.0	allergy	40.0	46.0	1.0	0.0	22.0
4	allergy	N	N	1.0	allergy	209.0	215.0	2.0	0.0	22.0
5	allert	PN	PN	1.0	alert	48.0	53.0	1.0	0.0	22.0

Figure 10. The Fetch Action Code Producing the Output from tpSpell

GENERATING A TERM-BY-DOCUMENT MATRIX

Once the text has been tokenized into the offset table, you use the tpAccumulate action to filter and reassign some of the terms, and to create a term-by-document weighted frequency table.

Filtering and reassigning the terms is done with the following options on the tpAccumlulate action:

- synonyms: Maps a set of terms to a canonical form of those terms and reduces the number of terms in your analysis.
- stopList: Eliminates specific terms from your analysis. A default stopList is provided in the reference library.
- reduce: Throws out infrequently occurring terms as these tend to be just noise in the collection.

In Figure 11 below, you can see how to create a custom synonym list to use as input to the tpAccumulate action using Python's StringIO function and SWAT's data message handler.

```
#Create Synonym List using Python's StringIO function
synonyms = StringIO('''term, termrole, parent, parentrole
"tsp", "N", "teaspoon", "N"
"tbsp", "N", "tablespoon", "N"
"carb", "N", "carbohydrate", "N"
"fridge", "N", "refrigerator", "N"
"hot chocolate", "nlpNounGroup", "hot cocoa", "nlpNounGroup"
"purchase", "V", "buy", "V"
"tummy", "N", "stomach", "N"
"tasty", "A", "delicious", "A"
"yummy", "A", "delicious", "A"
"yucky", "A", "disgusting", "A"
"mom", "N", "mother", "N"
"dad", "N", "father", "N"
"begin", "V", "start", "V"
"fast", "A", "quick", "A" ''')

#Add a handler to transport and load data into CAS using addTable action
handler = dmh.CSV(synonyms, skipinitialspace=True)
s.addtable(table='synonyms', replace=True, **handler.args.addtable)
```

Figure 11. An Example of Creating a Synonym List

Figure 12 shows the call to the tpAccumulate action. For your particular problem, you should consider experimenting with the different termWeight settings, the reduce= setting, and modify the terms on your stop and synonym lists to be useful for your data. Often the Mutual Information weighting, in conjunction with a target input is helpful, but in this case the setting seemed to cause overfitting, so it was not used.

```
#Running tpAccumulate action to generate term by document matrix
accumulate=s.tpAccumulate(cellWeight="LOG",
                          termWeight="entropy",
                          reduce=10,
                          offset={"name":"offset_spell"},
                          stopList={"name":"stop_list"},
                          synonyms={"name":"synonyms"},
                          terms={"name":"outterms", "replace":True},
                          parent={"name":"outparent", "replace":True},
                          complexTag=True)
```

Figure 12. The tpAccumulate Action Code

There are two primary outputs of the tpAccumulate action. The first is the terms table, which is a summary table containing the unique terms in the collection and the frequencies at which they occur. The second is the parent table, which is a compressed representation of the term-by-document weighted frequency table.

GENERATING DOCUMENT PROJECTIONS

In a term-by-document weighted frequency matrix, each document is represented with a vector whose length is equal to the number of distinct terms in the collection. While this is a numerical representation, it is too long and sparse to be useful, so your transformation of your input text to a numerical representation will be complete when the term-by-document frequency matrix is projected onto a smaller dimensional space. The tmSVD action in the textMining action set enables you to form this projection.

The action can do much more, such as discover topics in your data, but for your predictive model, you are primarily interested in the docPro table containing the k real-valued _Col1_-_Colk_ variables, where k is the number of dimensions you choose. These document projections variables, in conjunction with any other variables on your training data that you think might be useful, can be used as input when you train your predictive model.

The tmSVD action call shown in Figure 13 has several output tables. In addition to the docpPro table, the output scoreConfig table is the same parseConfig table you created with tpParse, together with additional information that tmSVD model needs at score time. The topics and termtopics output tables are not specifically required for the predictive model, but they are required for making the analytic store in the next subsection, so they are also requested. The option norm=″doc″ is specified to override the optimal topic calculation and instead focus on getting the best predictive ability. When you set the norm option in this way, you make sure that the document projections are normalized to unit length.

```
#Calling tmSvd action with K=100
svd=s.tmSvd(config={"name":"outConfig"},
            parent={"name":"outparent"},
            terms={"name":"outterms"},
            K=100,
            norm="doc",
            docPro={"name":"docPro_100", "replace":True},
            scoreConfig={"name":"scoreConfig_100", "replace":True},
            topics={"name":"outtopics_100", "replace":True},
            termTopics={"name":"termTopics_100", "replace":True})
```

Figure 13: The tmSvd Action Code

CREATING A TEXT MINING ANALYTIC STORE

The analytic store scoring mechanism has become a standard across SAS. The approach encapsulates needed information and data into a binary object, which is used for model deployment. In the tmAstore action, you create an analytic store by combining the content of several tables that were generated from the previous actions into a single analytic store table object. This table will be applied at score time with the score action from the analytic store action set.

Figure 14 contains the code to create the output analytic store table for the text analytics portion of your predictive model. In the following section you will see how to create a second analytic store for the SVM portion of your model.

```
#Running tmAstore action to generate Astore
text_astore=s.tmAstore(documents="amazonTraining",
                       docId="id",
                       text="text",
                       terms="outterms",
                       config="scoreConfig_100",
                       termTopics ="termTopics_100",
                       topics="outtopics_100",
                       saveState=s.CASTable("tmAstore_100", replace=True))
```

Figure 14. The tmAstore Action Code

BUILDING PREDICTIVE SUPPORT VECTOR MACHINES MODEL

The step above describes the final step of the text analytics training stages. The following stages defined from here are the predictive training stages. In this section, details on the analytical actions used for building the machine learning model will be described. As stated

in the introduction, the model chosen is SVM (Support Vector Machine). An SVM is a type of classifier that can learn from labeled training data and identify features that distinguish between different classes. For this project, the two classes that the SVM will classify are between reviews considered helpful and reviews considered unhelpful. To aid in the classification using an SVM, there are certain mathematical functions that are supported called kernels. Kernels are used to transform the data into a form that can make the data easier to classify. The purpose of these functions is to improve the separability in the data. Separability, in terms of SVM classification, is a property where two or more sets of data points can be easily divided into different classes. The functions transform the points so that a linear separator can be found that will make fewer errors.

Figure 15 below demonstrates the use of the svmTrain action, which uses linear and non-linear kernels to compute support vector machine (SVM) learning classifiers for the binary pattern recognition problem. The "degree" parameter specifies the degree of the polynomial kernel used. The "input" parameters specify the explanatory variables used for analysis. The document projections generated from the previous steps, as well as the review rating and review length are selected as these variables. The target variable, "helpful" is specified in the "target" parameter. When the action is run, the training results can be saved to a CAS table containing an analytic store and specified with the savestate option.

```
#SVM Model with All Input Variables (Doc Projections, Review Length, Star Rating)
svm_all=s.svmTrain(table= {"name":"docProJoin"},
                   degree=2, nominal = ["helpful", "score"],
                   inputs = ["_COL" + str(i+1) + "_" for i in range(100)] + ["score", "review_length"],
                   target = "helpful",
                   savestate = {"name":'svm_model_all', "replace":"true"},
                   id=["helpful"])
```

Figure 15. The svmTrain Action Code

MODEL SCORING

In this section you will see how to apply an analytic store scoring model. This standardized approach makes model deployment an easy step in many different contexts. Analytic stores are designed so that multiple different analytic stores can be applied one after another. In this case, you will first use the analytic store created from the text analytics component and then you will use the analytic store that encapsulates the SVM model. Figure 16 illustrates the model scoring process.

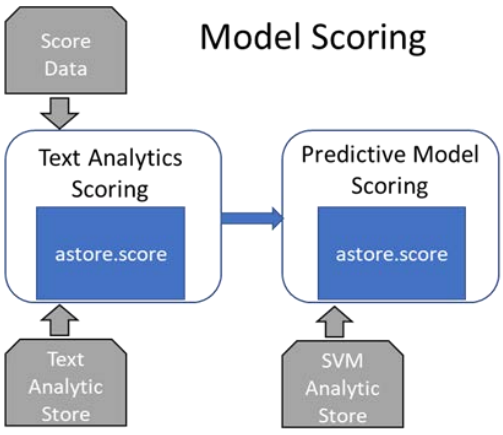

Figure 16. The Model Scoring Process

GENERATING SCORED DOCUMENT PROJECTIONS

You apply the text analytics scoring on the validation data with the score action. Your two inputs are the data you want to score and the text analytic store table. If there are additional variables that you want to use, then add them with the copyVars statement and they will be passed along together with the _Col1_-_Colk_ variables that you created at train time. The score action call is shown in Figure 17 and the output table created with the out option will then serve as input to the SVM scoring.

```
#Generate Scored Document Projections with astore.score
text_score=s.score(table="amazonValidation",
                   rstore="tmAstore_100",
                   out= s.CASTable("docproScore", replace=True),
                   copyVars={"id","score","text","helpful","review_length"})
```

Figure 17. Analytic Store Scoring Code for Text Model

SCORING THE SUPPORT VECTOR MACHINES PREDICTIVE MODEL

To score the SVM model, the output from the text scoring step and the SVM analytic store are submitted as input into the score action. The score action code for scoring the SVM model is shown in Figure 18. Your predictions will be included in the resulting output table.

```
#Score SVM Model - Doc Projections + Review Length + Star Rating
svm_score_all=s.score(
   table="docProScore",
   rstore="svm_model_all",
   out= s.CASTable("svm_model_allvar_score", replace=True)
   )
```

Figure 18. Analytic Store Scoring Code for SVM Model

ASSESSING RESULTS

To assess the results of your scored SVM model, you can use the assess action from the percentile action set. This action generates an output table with Receiver Operating Characteristic (ROC) information. Included in this output is the C statistic, which represents the area under the ROC curve, a common metric for assessing model performance. Figure 19 shows the code to generate the ROC information for your scored model.

```
#Assessing SVM Model - All Variables
assess=s.assess(table={"name":"svm_model_allvar_score"},
                inputs="p_helpful1",
                response="helpful", event="1",
                includeLift="false", rocOut={"name":"rocOut_all_var"})
```

Figure 19. Assess Action Code to Generate ROC Table

From the ROC output table, you will find that the area under the ROC curve for your model is 0.86 and that the overall misclassification at a p=0.50 cutoff is 18.7%. More specifically, the model successfully identified 11,079 of the 13,140 helpful reviews (84.3%) and 5,371 of the 7,100 unhelpful reviews (75.6%) in the validation data.

To further assess the model and to demonstrate the value that document projections were able to add, you can compare the predictive results with and without using the document

projections as a predictor. The misclassification results of three candidate models can be compared to determine which has the most predictive value:

- SVM Model using document projections, star rating, and review length
- SVM Model using only document projections (no star rating or review length)
- SVM Model using only star rating and review length

For the sake of space, the code to generate all three models and compare their misclassification rates is not included in this paper. Table 2 is a Jupyter Notebook output table created to compare the misclassification rates of the three models at a cutoff of p=0.50. This table shows that the model including all variables did the best job at classifying reviews as helpful.

Misclassification Rate Comparison

Variable Scenarios	Misclassification Rates
With all Variables	0.187253
Without Doc Projections	0.223370
With Only Doc Projections	0.239427

Table 2. Comparing Misclassification Rates

For a visual comparison between models, the Matplotlib library can be used to generate an ROC curve for each model. Figure 20 shows how to generate the curves.

```
plt.title('Receiver Operating Characteristic - Validation Dataset') #Give title of the plot

#Plot the values
plt.plot(fpf_all, tpf_all, label='All Variables (AUC=0.86)')
plt.plot(fpf_only_doc, tpf_only_doc, label='Only Doc Projections (AUC=0.81)')
plt.plot(fpf_no_doc, tpf_no_doc, label='No Doc Projections (AUC=0.78)')

#(x-axis value, y-axis value, design of plot line, labels to be given in the graph)

plt.legend(loc = 'lower right')#Specify location of the plot labels
plt.plot([0, 1], [0, 1],'r--')#Constructing the red dashed line as the 'benchmark' plot
plt.xlim([0, 1])#Specify x-axis limit
plt.ylim([0, 1])#Specify y-axis limit
plt.ylabel('True Positive Rate')#Specify y-axis label
plt.xlabel('False Positive Rate')#Specify the x-axis label
plt.show()#Print plot
```

Figure 20. Code to Generate ROC Curves

The resulting ROC curves are shown in Figure 21.

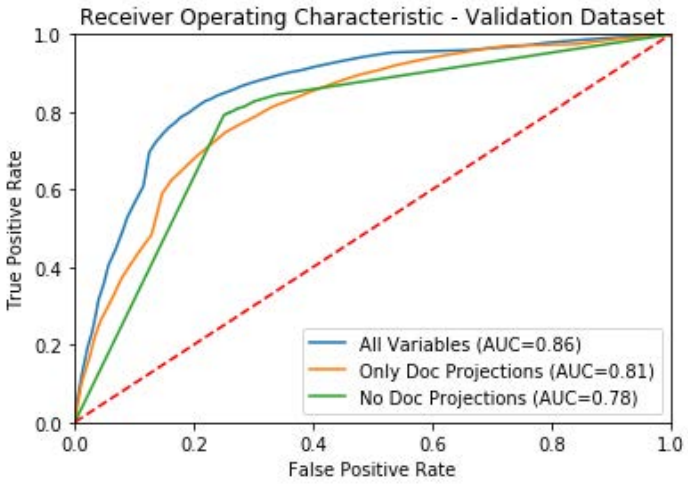

Figure 21. ROC Curves on Validation Data

The area under the curve improves significantly from 0.78 to 0.86 when document projections are included in the model. This demonstrates the value added through the text analytics model building process where unstructured text was converted into numeric input variables for use in the SVM model.

CONCLUSION

The SWAT library makes it easy for Python users to access and interact with the SAS Viya platform. Python users are able to write code in a familiar environment while gaining access to SAS text analytics and machine learning CAS actions. These actions were used throughout the course of this paper to highlight an approach for building an end-to-end text analytics model in SAS Viya using Python.

The value of incorporating unstructured text as input into a machine learning model was demonstrated. Amazon reviews were transformed into a numerical representation via the document projections output from the tmSvd action. Those document projections were combined with the star rating and review length as input into an SVM model to predict whether users will rate a review as being helpful. The results on the validation data proved the value of incorporating the document projections as they improved the misclassification rate and area under the curve significantly. The area under the curve from this model built with a traditional text mining approach plus an SVM can serve as a baseline for comparison against other techniques such as deep learning, an approach SAS makes accessible to Python users through the SAS Deep Learning Python (DLPy) package.

REFERENCES

Albright, R. 2004. SAS Institute white paper. "Taming Text with the SVD." ftp.sas.com/techsup/download/EMiner/TamingTextwiththeSVD.pdf.

Foreman, Carrie. 2019. "SWAT's it all about? SAS® Viya® for Python Users." *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3610-2019.pdf

Indelicato, Joe. 2019. "Open Visualization with SAS® Viya® and Python." *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3455-2019.pdf

SAS Institute Inc. 2020. "What's New in SAS Visual Text Analytics 8.5" *SAS® Visual Text Analytics 8.5: Programming Guide*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=casvtapg&docsetTarget=p0vouc3o8s7gq0n1p1b68jydb8id.htm&docsetVersion=8.5&locale=en

SAS Institute. 2020. "Support Vector Machine Action Set: svmTrain Action." *SAS® Visual Data Mining and Machine Learning 8.5: Programming Guide*. Cary, NC: SAS Institute Inc. https://documentation.sas.com/?docsetId=casactml&docsetVersion=8.5&docsetTarget=cas-svm-svmtrain.htm&locale=en

SAS Institute. 2020. "Percentile Action Set: assess Action." *SAS® Visual Analytics 8.5: Programming Guide*. Cary, NC: SAS Institute Inc. https://documentation.sas.com/?docsetId=casanpg&docsetTarget=cas-percentile-assess.htm&docsetVersion=8.5&locale=en

SAS Institute Inc. 2016. SWAT Documentation. https://developer.sas.com/apis/swat/python/v1.1.0/install.html

SAS Institute Inc. 2020. SAS Deep Learning Python (DLPy). https://github.com/sassoftware/python-dlpy

CONTACT INFORMATION

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

Nate Gilmore
Nate.Gilmore@sas.com

Vinay Ashokkumar
Vinay.Ashokkumar@sas.com

Russell Albright
Russell.Albright@sas.com