SAS® GLOBAL FORUM 2018

USERS PROGRAM

April 8 – 11 | Denver, CO

#SASGF

TEXT ANALYSIS ACCURACY AND EASE IN SAS® TEXT MINER VERSUS THE PYTHON NLTK SENTIMENT ANALYSIS PACKAGE

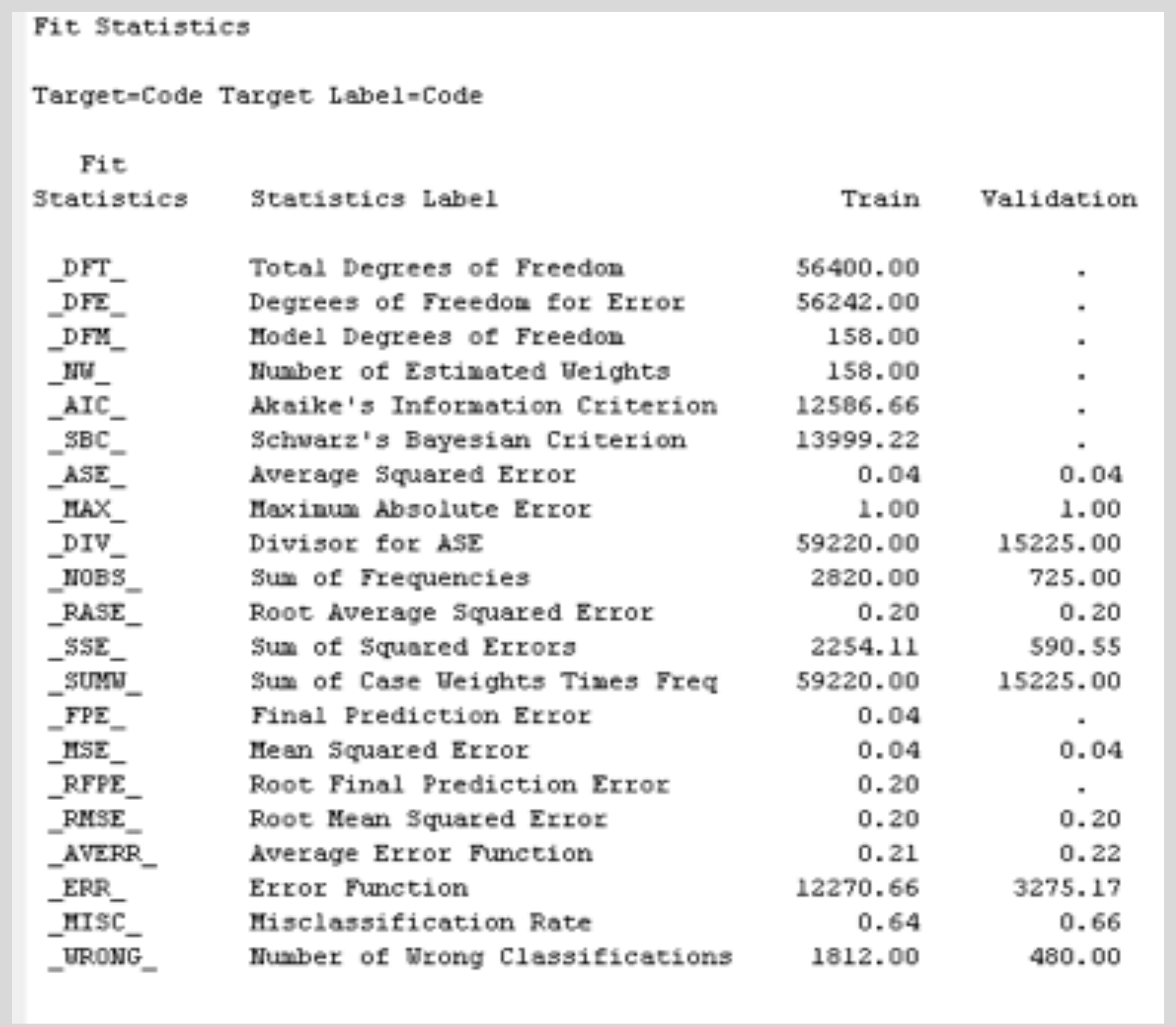
Jacob Braswell

ABSTRACT

- In higher education, institutions are constantly collecting data about their students' experiences at the university. Much of this data is in the form of free-form text responses. With text analytics, institutions are able to cut down the time spent analyzing this data by more than half and still have the same accuracy as they would if they analyzed it by reading and coding it manually.
- Can use text analytics to code through and analyze open-response survey data in both SAS® Text Miner and the Natural Language Toolkit (NLTK) in Python, and compare the two methods in regards to accuracy and user friendliness.
- We also discuss the other applications and benefits of using text analytics at institutions that need to access large amounts of information that is stored in the form of qualitative data both effectively and efficiently.

METHODS

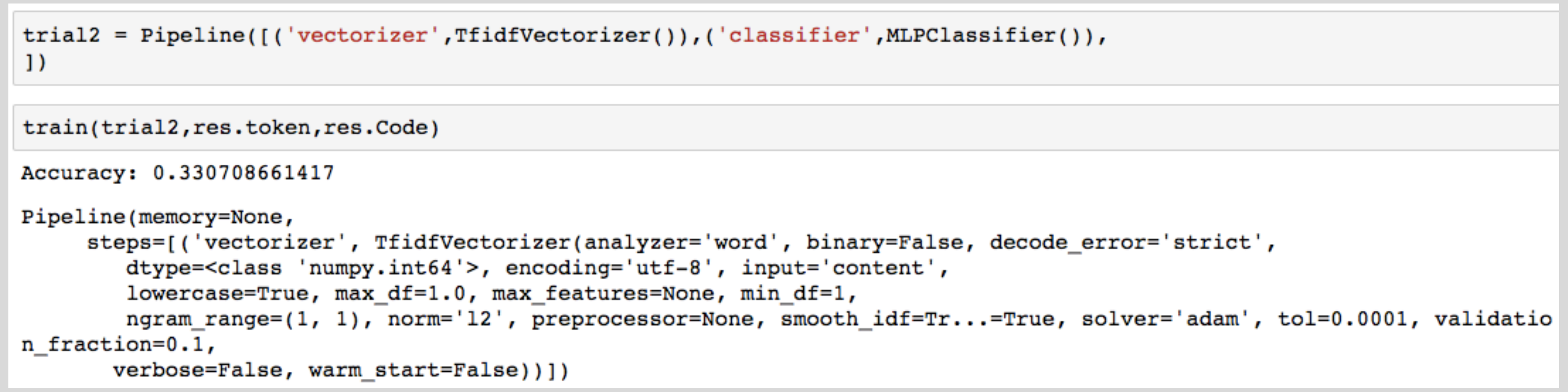
1. Load data set
2. Explore data set- how values are distributed, frequency count
3. Parse data to make analysis more accurate- for instance remove stop words, tokenize and remove punctuation
4. Convert text to vectors for run through of algorithm
5. Run through classifying function-Neural net
6. Output results and compare to other applied models
7. Check accuracy



```
Fit Statistics

Target=Code Target Label=Code

   Fit
Statistics    Statistics Label                       Train    Validation

 _DFT_        Total Degrees of Freedom            56400.00         .
 _DFE_        Degrees of Freedom for Error        56242.00         .
 _DFM_        Model Degrees of Freedom              158.00         .
 _NW_         Number of Estimated Weights           158.00         .
 _AIC_        Akaike's Information Criterion      12586.66         .
 _SBC_        Schwarz's Bayesian Criterion        13999.22         .
 _ASE_        Average Squared Error                   0.04      0.04
 _MAX_        Maximum Absolute Error                  1.00      1.00
 _DIV_        Divisor for ASE                     59220.00  15225.00
 _NOBS_       Sum of Frequencies                   2820.00    725.00
 _RASE_       Root Average Squared Error              0.20      0.20
 _SSE_        Sum of Squared Errors                2254.11    590.55
 _SUMW_       Sum of Case Weights Times Freq      59220.00  15225.00
 _FPE_        Final Prediction Error                  0.04         .
 _MSE_        Mean Squared Error                      0.04      0.04
 _RFPE_       Root Final Prediction Error             0.20         .
 _RMSE_       Root Mean Squared Error                 0.20      0.20
 _AVERR_      Average Error Function                  0.21      0.22
 _ERR_        Error Function                      12270.66   3275.17
 _MISC_       Misclassification Rate                  0.64      0.66
 _WRONG_      Number of Wrong Classifications      1812.00    480.00
```

RESULTS

- From the output of our results we find that our standard neural network in Python is around 33% accurate.
- Our SAS neural network is about 34% accurate.
- These are both not super effective but for a multiple category classification done with a simple neural network it is still rather effective considering there are still many other aspects to control and utilize in our analysis



```
trial2 = Pipeline([('vectorizer',TfidfVectorizer()),('classifier',MLPClassifier()),
])
```

```
train(trial2,res.token,res.Code)
```

```
Accuracy: 0.330708661417

Pipeline(memory=None,
     steps=[('vectorizer', TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=Tr...=True, solver='adam', tol=0.0001, validatio
n_fraction=0.1,
       verbose=False, warm_start=False))])
```

CONCLUSION

- Though you have more control over how and what you can implement when doing text analytics in python it is more difficult to parse the data and prepare it for analysis in any algorithm that you may use.
- SAS Text Miner is easier to use as well as accurate and is far more user friendly with explanations regarding each node and algorithm.
- In terms of accuracy both are about the same, which makes sense since we used similar methods.

REFERENCES

- Perkins, J. (2010). *Python text processing with NLTK 2.0 Cookbook: Over 80 practical recipes for using Python's NLTK suite of libraries to maximize your natural language processing capabilities*. Birmingham: PACKT Publishing.

SAS® GLOBAL FORUM 2018

April 8 – 11 | Denver, CO
Colorado Convention Center

#SASGF

Paper 2519-2018

TEXT ANALYSIS ACCURACY AND EASE IN SAS® TEXT MINER VERSUS THE PYTHON NLTK SENTIMENT ANALYSIS PACKAGE

Jacob Braswell, Brigham Young University

ABSTRACT

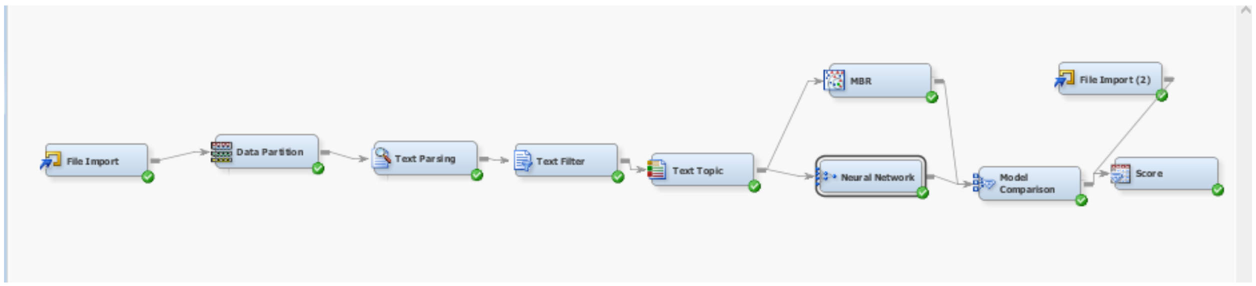
With machine learning conquering many facets of data analysis, jobs that used to be time consuming are now streamlined into simple tasks. One aspect is text analytics. With many companies receiving thousands of open-response complaints daily, text analytics helps companies know exactly what their customers need and how to best address those needs without spending hours reading through each individual response. In higher education, institutions are constantly collecting data about their students' experiences at the university. Much of this data is in the form of free-form text responses. With text analytics, institutions are able to cut down the time spent analyzing this data by more than half and still have the same accuracy as they would if they analyzed it by reading and coding it manually. In this paper, we discuss the process of using text analytics to code through and analyze open-response survey data in both SAS® Text Miner and the Natural Language Toolkit (NLTK) in Python, and compares the two methods in regards to accuracy and user friendliness. We also discuss the other applications and benefits of using text analytics at institutions that need to access large amounts of information that is stored in the form of qualitative data both effectively and efficiently.

INTRODUCTION

The data that we encounter daily comes in many forms and the technology that we currently use do an amazing job processing and analyzing that data very easily in most cases. One exception is in analyzing and recording text data, where analyzing sentiment or even categorizing topics into more than two categories is still a realm that remains challenging. In this paper the use of both SAS Text Miner and pythons NLTK library is discussed in its use of helping to analyze open response data and to categorize it using a simple neural network. From this paper the reader will better understand where basic text analytics and categorization is currently at and where there is still room for improvement. The dataset we use is collected on students asking them what they feel like can be improved at school and the targets are various categories such as quality of faculty or course structure.

ANALYSIS IN SAS

The SAS Text Miner interface consists of nodes that have various functions from Modify, Asses, and Text Mining to name a few. With this set up it makes for easy navigation and allows the user to see the direction of the model step by step. In our case we mainly use the Text Mining, Sample, Model and Assess functions. Within each of these sections there are different nodes that can be used to help accomplish the goals of the analysis. In our case we are going to use the nodes shown in the below figure.

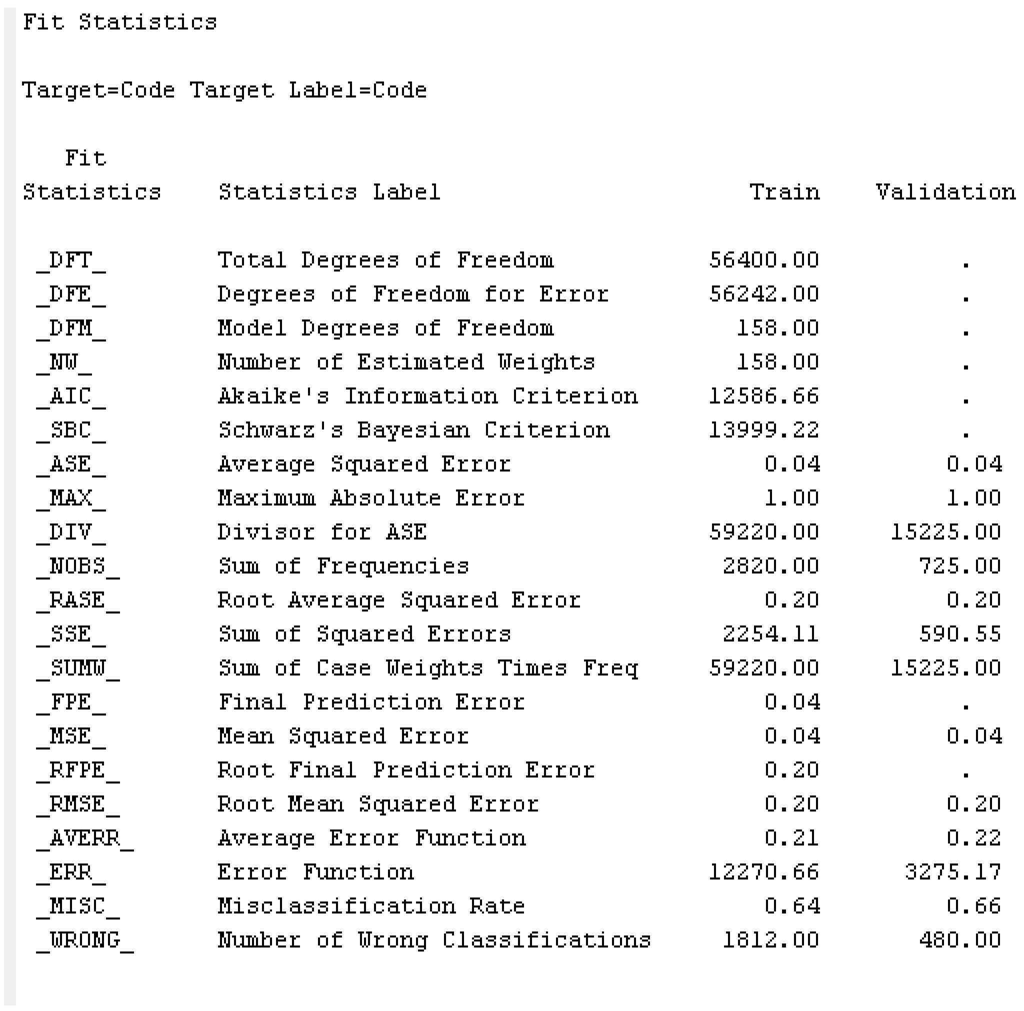


Display 1. Model Used to Analyze the Text in SAS

In the above figure we see the form of our analysis and how we used the nodes within the Miner environment to accomplish our goals of analysis. First we read in the data and separate our data into 90% training and 10% testing, so that we can test the accuracy of our predictions later on. Then we begin using the Text Mining group of nodes to make our data more easily categorized in our algorithm. First we want to parse our text data, then filter and then link our text to certain topics, each of these nodes allows us to normalize into a more readable mode in order to be put through our algorithm. For instance, in the text parsing node we can get a feel for the frequency of text, see what the different words and phrases' roles in sentences are in our responses, tokenize words, and can set a limit for how frequent a word needs to be for it to be included in the analysis that is run by our algorithm. From there we filter out the stop words and any other parts of our data that may be hinder some to our analysis. In the Text Topic node, we connect documents and certain categories together as gathered from our training dataset. Then we run the filtered data through a neural network and a nearest k means algorithm in the MBR node for comparison to the accuracy of our neural network node. In each of these nodes, after you have run them you can look through at the results tab which includes various information about how the node ran and other various metrics that help you measure the outcome of how the node ran. This is particularly helpful in making the process of text analytics user friendly and allows the user to have better access to any metrics that could possibly be needed in model evaluation.

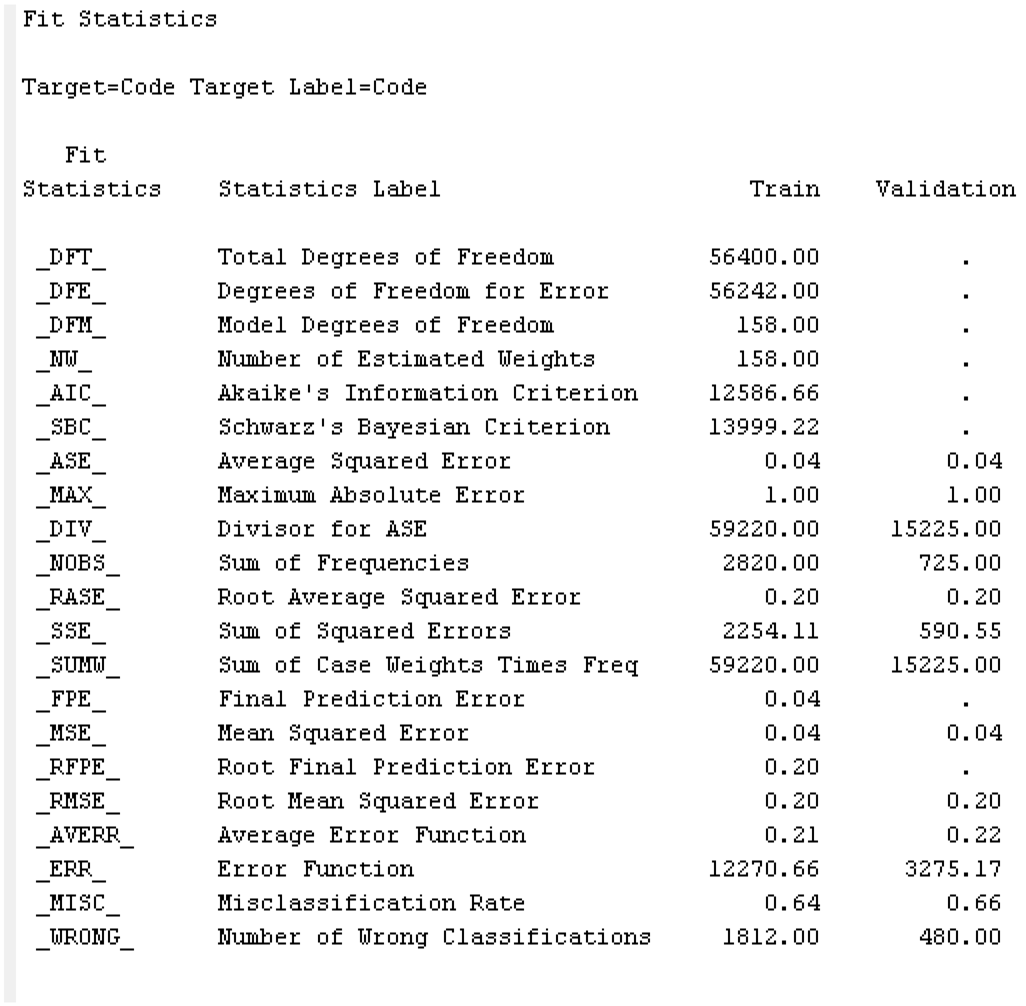
RESULTS

After running our model through the above model we go to the various nodes in our model and examine the results to see the distribution of the dataset that we ran through our trained model at each point. More particularly if we are wanting to see how accurate our neural network did in the training and validation steps we can simply click on that node and examine the results by clicking on the results button. When we do this we find, among other things, the below table that shows us the misclassification rate of our neural network in scoring our text given the target. We find from our table that the misclassification rate of our model is .64 in the training phase and .66 in the validation phase. This can be interpreted as around 36% of the time in training our neural network correctly classifies a text to the correct target and 34% of the time our neural network correctly classifies a text to the correct target when we are validating the data. Though these numbers aren't particularly amazing we know that the analysis that we performed is rather basic and that the data set that we have is rather messy multi categorical multi classified data. So we can be satisfied that even on a hard data set such as this we are still getting pretty decent results given how hard our data set is to classify. Also in regards to ease of the process, once we understand the basic structure of text analytics and understand how the classification nodes work we can implement to solve our problem at hand making the learning curve for working on a problem such as this rather easy and straightforward.



```
Fit Statistics

Target=Code Target Label=Code

   Fit
Statistics    Statistics Label                       Train    Validation

 _DFT_        Total Degrees of Freedom            56400.00          .
 _DFE_        Degrees of Freedom for Error        56242.00          .
 _DFM_        Model Degrees of Freedom              158.00          .
 _NW_         Number of Estimated Weights           158.00          .
 _AIC_        Akaike's Information Criterion      12586.66          .
 _SBC_        Schwarz's Bayesian Criterion        13999.22          .
 _ASE_        Average Squared Error                   0.04        0.04
 _MAX_        Maximum Absolute Error                  1.00        1.00
 _DIV_        Divisor for ASE                     59220.00    15225.00
 _NOBS_       Sum of Frequencies                   2820.00      725.00
 _RASE_       Root Average Squared Error              0.20        0.20
 _SSE_        Sum of Squared Errors                2254.11      590.55
 _SUMW_       Sum of Case Weights Times Freq      59220.00    15225.00
 _FPE_        Final Prediction Error                  0.04          .
 _MSE_        Mean Squared Error                      0.04        0.04
 _RFPE_       Root Final Prediction Error             0.20          .
 _RMSE_       Root Mean Squared Error                 0.20        0.20
 _AVERR_      Average Error Function                  0.21        0.22
 _ERR_        Error Function                      12270.66     3275.17
 _MISC_       Misclassification Rate                  0.64        0.66
 _WRONG_      Number of Wrong Classifications      1812.00      480.00
```

Table 1



```
Fit Statistics

Target=Code Target Label=Code

   Fit
Statistics    Statistics Label                     Train    Validation

 _DFT_        Total Degrees of Freedom          56400.00         .
 _DFE_        Degrees of Freedom for Error      56242.00         .
 _DFM_        Model Degrees of Freedom            158.00         .
 _NW_         Number of Estimated Weights         158.00         .
 _AIC_        Akaike's Information Criterion    12586.66         .
 _SBC_        Schwarz's Bayesian Criterion      13999.22         .
 _ASE_        Average Squared Error                 0.04        0.04
 _MAX_        Maximum Absolute Error                1.00        1.00
 _DIV_        Divisor for ASE                   59220.00    15225.00
 _NOBS_       Sum of Frequencies                 2820.00      725.00
 _RASE_       Root Average Squared Error            0.20        0.20
 _SSE_        Sum of Squared Errors              2254.11      590.55
 _SUMW_       Sum of Case Weights Times Freq    59220.00    15225.00
 _FPE_        Final Prediction Error                0.04         .
 _MSE_        Mean Squared Error                    0.04        0.04
 _RFPE_       Root Final Prediction Error           0.20         .
 _RMSE_       Root Mean Squared Error               0.20        0.20
 _AVERR_      Average Error Function                0.21        0.22
 _ERR_        Error Function                    12270.66     3275.17
 _MISC_       Misclassification Rate                0.64        0.66
 _WRONG_      Number of Wrong Classifications    1812.00      480.00
```

Table 1. Class Variable Summary Statistics for Analysis

ANALYSIS IN PYTHON

With NLTK there are many functions that we can use to help us in our analysis of the data set. With the case in all datasets there is a lot of parts of the data that are outliers and have the ability to make our accuracy a lot lower than we would expect if our data had already been cleaned. Some of the most commonly used features are tokenizing and removing stop words from the text data set. NLTK has features to help us to accomplish this. Stop words are words such as “the”, “as”, “in” etc. Through removing these types of words we are better able to improve the performance of our model. Tokenizing is another tool that we can implement to help improve the performance of our algorithm. Tokenizing free text separates each individual word either according to word or sentence, in our case we separated it according to word since most of our text was only a few sentences per response. We also removed punctuation to avoid having any other hindrance in our model’s performance. From there we convert our words to vectors and run it through our our Neural Network.

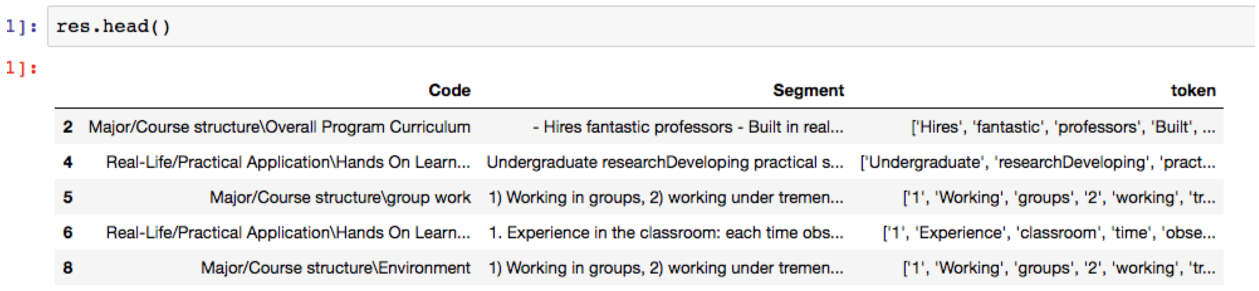
One thing that could have been implemented to improve the accuracy of our neural network would be using stemming, where we take words and cut it down to its stem. For instance, we can make going, gone, and go all into the stem go to help the neural network classify our words correctly.

Below is the code that is used to remove stopwords, punctuation, and to tokenize the responses in our dataset:

from nltk.corpus import stopwords

```
from nltk.tokenize import word_tokenize
stop_words = set(stopwords.words("english"))
res['Segment'] = res.Segment.astype(str)
res.dropna(axis=0,how='all')
responses = res['Segment']
import string
responses = responses.apply(lambda x:''.join([i for i in x
                                      if i not in string.punctuation]))
res["token"] = responses.fillna("").map(nltk.word_tokenize)
res['token'] = res['token'].apply(lambda x: [item for item in x if item not in stop_words])
```

Display is sample display or screen capture.



```
1]: res.head()
```

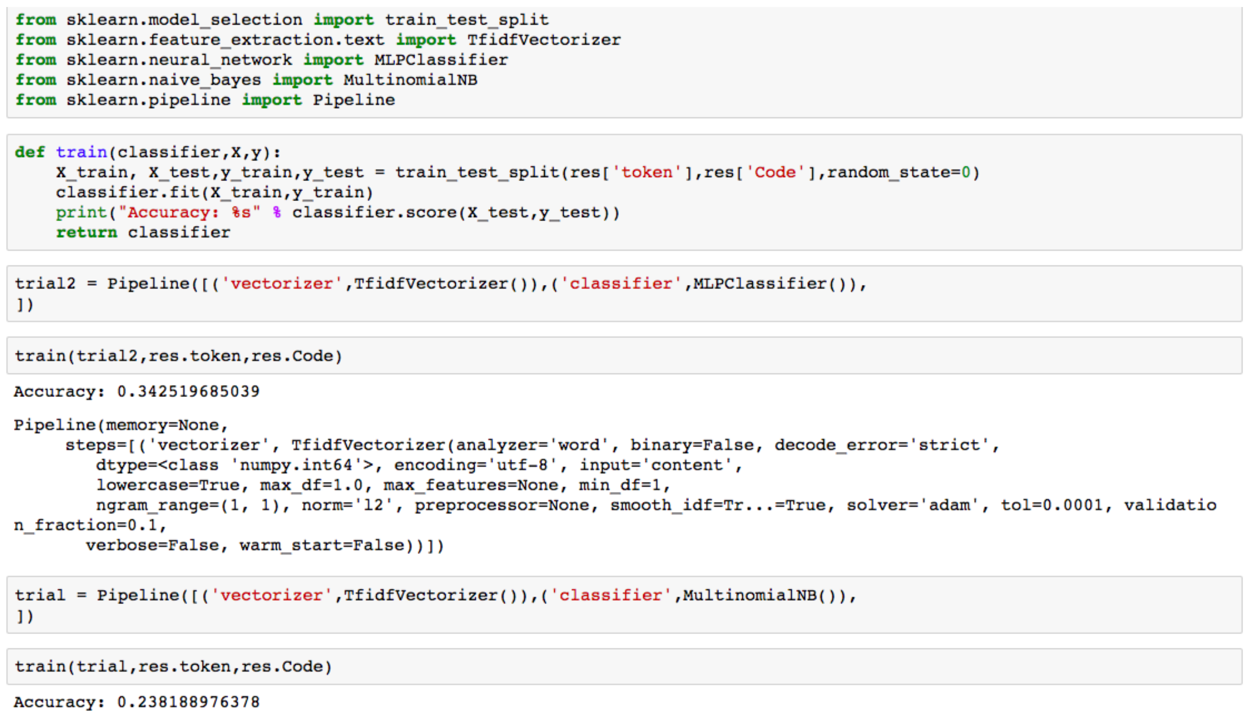
```
1]:
```

	Code	Segment	token
2	Major/Course structure\Overall Program Curriculum	- Hires fantastic professors - Built in real...	['Hires', 'fantastic', 'professors', 'Built', ...
4	Real-Life/Practical Application\Hands On Learn...	Undergraduate researchDeveloping practical s...	['Undergraduate', 'researchDeveloping', 'pract...
5	Major/Course structure\group work	1) Working in groups, 2) working under tremen...	['1', 'Working', 'groups', '2', 'working', 'tr...
6	Real-Life/Practical Application\Hands On Learn...	1. Experience in the classroom: each time obs...	['1', 'Experience', 'classroom', 'time', 'obse...
8	Major/Course structure\Environment	1) Working in groups, 2) working under tremen...	['1', 'Working', 'groups', '2', 'working', 'tr...

Display 2. Head of data frame after implementing above code

RESULTS

After running our data through our various filters that may prevent the text from being less easily analyzed, we convert our numbers to vectors using the SciKitLearn Library's TidifVectorizer function to convert our text into vectors of numbers in order for it to pass through our neural network as a one dimensional array. There are many other ways to accomplish this task such as glove vectorizing but this one is the most basic ways to accomplish this task. From there we run the vectors through our neural network and output our results. As we see from the display below, we get an accuracy score of around .34.



```
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.neural_network import MLPClassifier
from sklearn.naive_bayes import MultinomialNB
from sklearn.pipeline import Pipeline
```

```
def train(classifier,X,y):
    X_train, X_test,y_train,y_test = train_test_split(res['token'],res['Code'],random_state=0)
    classifier.fit(X_train,y_train)
    print("Accuracy: %s" % classifier.score(X_test,y_test))
    return classifier
```

```
trial2 = Pipeline([('vectorizer',TfidfVectorizer()),('classifier',MLPClassifier()),
])
```

```
train(trial2,res.token,res.Code)
```

```
Accuracy: 0.342519685039

Pipeline(memory=None,
     steps=[('vectorizer', TfidfVectorizer(analyzer='word', binary=False, decode_error='strict',
        dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
        lowercase=True, max_df=1.0, max_features=None, min_df=1,
        ngram_range=(1, 1), norm='l2', preprocessor=None, smooth_idf=Tr...=True, solver='adam', tol=0.0001, validatio
n_fraction=0.1,
       verbose=False, warm_start=False))])
```

```
trial = Pipeline([('vectorizer',TfidfVectorizer()),('classifier',MultinomialNB()),
])
```

```
train(trial,res.token,res.Code)
```

```
Accuracy: 0.238188976378
```

Display 3. Results output by Python

This result may seem low but later on we compare the accuracy of our neural network to a multinomial NaïveBayes classifier and we get an accuracy of around .23. Keeping this in mind we see that, although our result has room for improvement, is still somewhat accurate considering we are using a simple neural network with only a few variables controlled in the pre training process. In regards to user friendliness, NLTK and Python's system often make it hard to deal with the way data is read in in regards to type, making it hard to implement some of the features that are available in the NLTK library.

CONCLUSION

From looking at the two different ways analysis we see that the SAS Text Miner method produces around the same results as our python analysis, this makes sense due to the fact that we used relatively the same methods to analyze the data. the SAS interface is a more user friendly method due to the fact that most of the steps in pre-analyzing are very controlled and there is a simple click and point interface. With Python's interface you do have to have a more solid grasp of what each of your steps are in need of and how to best implement those steps on the data type that you have. This of course is a good thing if you are wanting to do a deep analysis and have specific things that you wish to analyze. Since we are simply wanting to do a quick analysis of our data and attempt to categorize the data from our model we created this becomes a bit of a hindrance with the many different options. Python is also not limited to just one package or method which allows you to go through and customize the process of text analysis, whereas SAS does give you plenty of options but if your method isn't one of the options you are hard pressed to find it. Regardless of this, SAS's Text Miner proves to be the better option in our case for analyzing our dataset.

REFERENCES

1. Perkins, J. (2010). *Python text processing with NLTK 2.0 Cookbook: Over 80 practical recipes for using Python's NLTK suite of libraries to maximize your natural language processing capabilities*. Birmingham: PACKT Publishing.

CONTACT INFORMATION

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

Jacob Braswell
Brigham Young University
jacob.braswell00@gmail.com