Speaker Classification on General Conference Talks and BYU Speeches

Cason Wight

January 8, 2021

Abstract

The leaders of the Church of Jesus Christ of Latter-Day Saints have distinct ways of speaking. This project scrapes text data from General Conference talks and BYU speeches to build classification models on authorship from quotes. Term frequency-inverse document frequency is used for quote-to-matrix conversion. Both multinomial logistic regression and support vector machines are used as classification techniques. For speakers with more quotes available, support vector machines made more accurate predictions, and for speakers with fewer available quotes, multinomial logistic regression had higher classification rates. Neither method did particularly well in attributing quotes (highest accuracy of 24% for a speaker that makes up 14% of the texts). Predicting if a given talk from President Oaks is from 2020 or from earlier is also challenging. The multinomial logistic regression performed better at this task, but correctly classified only 7% of the 2020 talks.

1 Introduction

For most students at BYU, the most influential speakers in their lives are the leaders of the Church of Jesus Christ of Latter-Day Saints. The leadership of the Church comprises of many different groups. The foremost of these groups are the First Presidency (three members, including the President of the Church) and the Quorum of the Twelve Apostles. All 15 of these men address the church at least semi-annually in a global broadcast called *General Conference*. Additionally, BYU holds regular campus-wide broadcasts called *BYU Speeches*. BYU invites many different people to address the university and regularly hosts one of these 15 prophets of the Church. For many, the messages shared by these men become fundamental teachings on how to live. Different speakers are favorites to different listeners.

Every person speaks in a unique way. The background, interests, age, and context of a speaker will influence the words they use. Specific words, speech patterns, and can distinguish speakers from each other. Researchers have suggested that people have *linguistic fingerprints*, meaning people's word choices and language styles are distinct enough for author/speaker identification. Eder (2011) notes that in addition to the frequency of specific words/phrases other style markers are helpful in author identification, although less useful in English, when compared to other languages.

The field of breaking down language into statistical and machine learning analysis is known as *natural language processing* (NLP). Authorship attribution is only one of many NLP problems. NLP is also used in other types of text classification, which can be done in many ways. Some of the most popular methods of classification include regression, support vector machines (SVMs), classification trees, or deep learning. Often times, the features selected from the text dictate which classification methodology is preferable. Sentiment analysis, summarization, and market intelligence are other avenues where NLP can also be helpful.

The main purpose of this project is to perform authorship classification on the 15 leaders of the Church mentioned above, using all addresses from general conference and BYU speeches. In the last year, leaders of the church, especially President Dallin H. Oaks, have directly addressed issues such as political unrest and racism. Some have noted that this is uncommon for President Oaks. In the Salt Lake Tribune, Stack et al. (2020) said that in giving such talks, the leaders are "breaking with the more general approaches of the past." In addition to predicting the speakers, this project will also predict if a quote from President Oaks comes from this year (2020) or not.

2 Data

There is no available data set that contains all of the addresses of the 15 leaders of the church mentioned. Instead, all talks available on the Church's website and the BYU Speeches website were webscraped and cleaned for the purpose of this project. The intention is to predict the author from a single quote, so the talks were split by paragraph, resulting in a set of over 35,000 total quotes from the 15 men mentioned; the earliest address is from 1973 and the most recent from 2020. After removing quotes with fewer than 30 characters, $34,066$ texts remained.

Classification methods require conversion of text to numerical representations. One such representation could be style markers. The use of different rhetorical language choices are often recorded numerically, see Collins (2003), for example. This project will instead tokenize the paragraphs based on the words and phrases used in each quote, using n-grams. These are counts for each possible sequence of n words, for all paragraphs (after removing stop words). For example, tokenizing the sentence "NLP is difficult without a computer" with 2-grams, would yield a vector giving 1s for

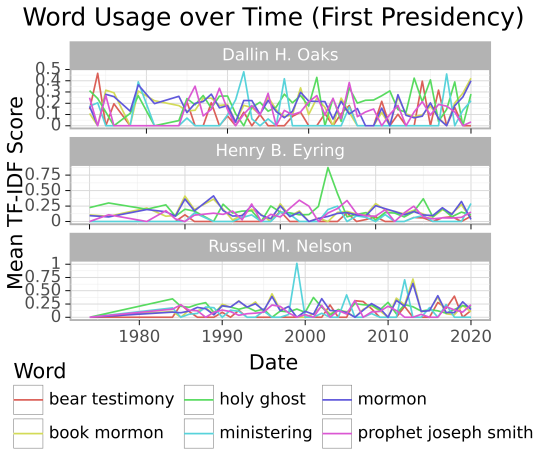

Figure 1: Different word usage in First Presidency over time.

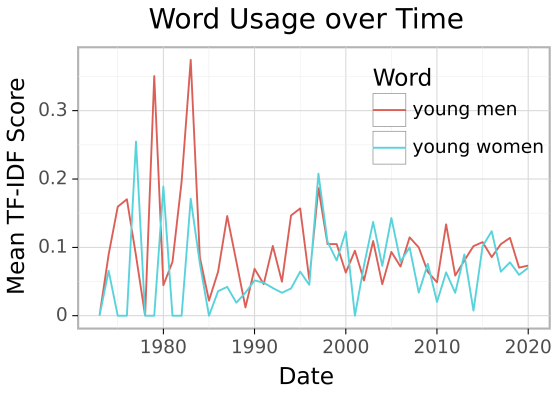

Figure 2: Use of "young men" and "young women" in the current 15 leaders of the Church.

"nlp difficult", "difficult without", and "without computer". Notice that sentences are stripped of punctuation and filler words. Kruczek et al. (2020) gives a good introduction to n-grams and their benefits. Among the paragraphs in this data set, 2,640,990 possible words, 2-grams, 3-grams, 4-grams, and 5-grams were present. For every quote, the counts for each of these was phrases were used as the initial numerical representation for the paragraphs.

For classification, the raw counts for words/phrases are less useful than the relative uniqueness of words used by a speaker. One way to approach this style of metric is to use the term frequency-inverse document frequency (TF-IDF). For a thorough review of this metric, see Shi et al. (2009). The intuitive understanding of this approach is that it measures how much a given phrase is used in a particular text, and multiplies by the log of the inverse of how often the phrase appears among all texts. Thus if a word is used infrequently among other texts, but frequently in a particular text (making it unique to the specific text), the TF-IDF score will be high. If a phrase is common among all texts, the corresponding TF-IDF scores will be low. Formally, TF-IDF scores are calculated by $w_{i,j} = \frac{\#\text{ of times term } j \text{ appears in text } i}{\#\text{ of terms in text } i} \times \log\left(\frac{\#\text{ of texts}}{\#\text{ of texts with term } j}\right)$. Of the millions of possible phrases tokenized, only the 2,500 with the highest mean TF-IDF score are used in this project, for computational feasibility.

The TF-IDF scores can show some interesting trends in the data. First of all, the different speakers do, in fact, favor different words over each other. It is also notable that these preferences seem to change over time as well. Figure 1 shows how the three members of the first presidency use a set of arbitrary phrases in different amounts over time. Figure 2 shows how the current 15 leaders have used the phrase "young men" and "young women" over the years as well. This graph shows that the mean TF-IDF score was much higher for young men in the 1970s, 1980s, and 1990s. In the 2000s and 2010s, the scores for the two different phrases have balanced. Notice that these are not counts themselves, but TF-IDF scores, so they are phrase counts within a text, down-weighted by the phrase's frequency among all texts. Although interesting (and most likely helpful in predictions), the year of each quote will be ignored in prediction, as it does not align with the purpose of this project.

The result of the feature selections is a $34{,}066 \times 2{,}500$ matrix $\mathbf{X}$. The corresponding response $\mathbf{y}$ is a $34{,}066 \times 1$ column vector containing the corresponding speaker for each quote. There are several methods for classification analysis. This project will use multinomial logistic regression (MLR) and support vector machines (SVMs). Naive bayes or deep learning methodology would also be appropriate in this context, but is beyond scope of this project.

3 Methodology Multinomial logistic regression is a generalization of binary logistic regression to the case of multiple groups. In MLR, the log of the odds of being in a given class j is compared to some base class J (with a total of J classes). Thus if $\pi_{i,j} = \Pr(y_i = \text{speaker}_j)$ is the probability of observation i belonging to group j, the regression model is as follows:

$$\ln\left(\frac{\pi_{i,j}}{\pi_{i,J}}\right) = \alpha_j + \mathbf{x}_i'\boldsymbol{\beta}_j, \qquad \forall\ j \in \{1, \ldots, J-1\} \tag{1}$$

The $\boldsymbol{\beta}_j$ is the vector of regression coefficients for the effect of each column on the log-odds of being in class j over the base class J. In this case, $\boldsymbol{\beta}_j = \begin{bmatrix}\beta_{j,1} & \beta_{j,2}, \ldots, \beta_{j,K}\end{bmatrix}'$ is the vector of effects of each of the 2,500 TF-IDF scores of each observation on the log-odds of being in class j. The $J-1$ instances of equation 1 can be converted from log-odds to probabilities through the following:

$$\pi_{i,j} = \frac{\exp\left\{\alpha_j + \mathbf{x}_i'\boldsymbol{\beta}_j\right\}}{\sum_{m=1}^{J}\exp\left\{\alpha_m + \mathbf{x}_i'\boldsymbol{\beta}_m\right\}}, \qquad \forall\ j \in \{1, \ldots, J\} \tag{2}$$

A simple way (used in this project) to convert the probabilities to predictions is that the class j that has the highest probability, $\underset{j\in\{1,\ldots,J\}}{\arg\max}(\pi_{i,j})$, is the predicted class for observation i. Variable selection can be difficult, so lasso regularization is used in this project instead. Typically, regression maximizes a likelihood function. Lasso regularization instead maximizes the following:

$$\mathcal{L}(\boldsymbol{\beta}) + \text{penalty} = \prod_{i=1}^{N}\prod_{j=1}^{J}\pi_{i,j} + \lambda\sum_{i=1}^{N}\sum_{k=1}^{K}|\beta_{j,k}| \tag{3}$$

The lasso regularization will shrink smaller effects (and those with poor predictive ability) to zero, thus automatically performing the variable selection on the $2{,}500 \times 15 = 37{,}500$ possible $\beta_{j,k}$s.

The intuitive understanding of SVMs is that hyperplanes are drawn to separate groups. The two-group case with only two variables (x_1, x_2) is simple to understand, but the general case will subsequently be explained. The first step is to center and scale the data so that each column has mean 0 and standard deviation 1. Let the discrete response be 1 for group 1 and -1 for group 2. In this situation (with two groups and two variables), suppose points are easily separable, where a line, $x_1 = mx_2 + b$, could be drawn to perfectly separate the two groups (as in Figure 3). This means that for all observations i in group 1, $x_{1,i} > mx_{2,i} + b$, and for all observations i in group 2, $x_{1,i} < mx_{2,i} + b$. The concept of SVM is built up from this simple case. This simple problem does not change when considering only the "border" points, as opposed to all of the points. These reference points are row vectors and are the only points required to define a discriminating line, thus they are called *support vectors*.

If a perfectly discriminating line exists, then there are infinite lines that could be drawn; in the working example of Figure 3, any line between the two dashed lines could perfectly discriminate the groups. The discriminant line could also be defined as $0 = \beta_0 + \mathbf{x}_0'\boldsymbol{\beta}$. The distance between a vector $\mathbf{x}_0$ and a hyperplane is $\frac{\beta_0 + \mathbf{x}_0'\boldsymbol{\beta}}{||\boldsymbol{\beta}||}$. The smallest distance from the support vectors to the classifier hyperplane is called the *margin* $M = \min_i \left[y_i(\beta_0 + \mathbf{x}_i'\boldsymbol{\beta})\left[||\boldsymbol{\beta}||^{-1}\right]\right]$. A support vector classifier uses the β_0 and $\boldsymbol{\beta}$ that maximize the margin. Formally this is the hyperplane $(\beta_0, \boldsymbol{\beta})$ solved by the following: $\underset{\beta_0,\boldsymbol{\beta}}{\arg\max} M$, with the constraint that $y_i(\beta_0 + \mathbf{x}_i'\boldsymbol{\beta})\left[||\boldsymbol{\beta}||^{-1} \geq M\ \forall\ i\right.$. Predictions from this method become $\hat{y}_0 = \text{sign}(\beta_0 + \mathbf{x}_0'\boldsymbol{\beta})$. The function $f(\mathbf{x}_0) = \hat{\beta}_0 + \mathbf{x}_0'\hat{\boldsymbol{\beta}}$ is called the *classifier*.

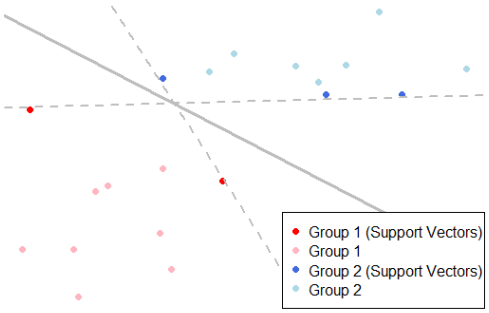

Figure 3: Support vector classification.

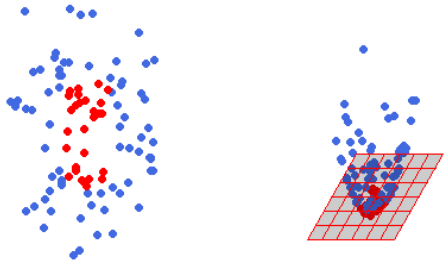

Figure 4: Kernel function in SVM.

The scale of $\boldsymbol{\beta}$ is not relevant, so it could be set such that $M = ||\boldsymbol{\beta}||$, and the problem of solving for the classifier becomes $\arg\min_{\beta_0,\boldsymbol{\beta}} \frac{1}{2}||\boldsymbol{\beta}||^2$, constrained by $y_i(\beta_0 + \mathbf{x}_i'\boldsymbol{\beta}) \geq 1$. From here, further mathematical simplifications have also been shown.

With real data, a hyperplane that perfectly separates groups typically does not exist. One way to account for this is the addition of a slack variable ϵ_i. For every vector $\mathbf{x}_i$, ϵ_i is a measurement of how close a vector is to the hyperplane. This value ranges from 0 at the margin to 1 at the hyperplane, and ≥ 1 if the point is on the "wrong" side of the hyperplane. Incorporating the ϵ_is, it can be shown that the previous solution then becomes $\arg\min_{\beta_0,\boldsymbol{\beta}} \frac{1}{2}||\boldsymbol{\beta}||^2$, constrained by $y_i(\beta_0 + \mathbf{x}_i'\boldsymbol{\beta}) \geq 1 - \epsilon_i$ and $\sum_{i=1}^{N} \epsilon_i \leq B$. In this formula, B is the tuning parameter. The greater B is, the more misclassification is allowed when finding the hyperplane. This parameter is tuned through cross validation (prediction accuracy on holdout data, testing many different potential values for B). It can be shown that this classifier finally simplifies to $f(\mathbf{x}_0) = \hat{\beta}_0 + \sum_{i:y_i(\beta_0+\mathbf{x}_i'\boldsymbol{\beta})>1-\epsilon_i} \hat{\lambda} y_i \mathbf{x}_0'\mathbf{x}_i$, which depends only on dot products of the support vectors.

Another problem that arises in SVM is that many times, a hyperplane on the raw data simply has no useful way to "slice" the observations. For example, in Figure 4, there is no line that could be drawn to separate one group from the other, although the separation seems clear. One strength of the dot-product definition of the classifier is that basis function expansions could easily be incorporated on the linear function. If a set of points are expanded into a higher dimension, a hyperplane may be able to separate groups in the higher dimension. In Figure 4, the (x, y) vectors are transformed into $(x, y, x^2 + y^2)$ vectors; the result is then a "bowl" shape, where a 2-d plane could easily separate the two groups. There are many computationally efficient expansions (*kernels*) h that are used in SVM. The solution to the classifier, when incorporating a kernel h becomes $f(\mathbf{x}_0) = \hat{\beta}_0 + \sum_{i:y_i(\beta_0+\mathbf{x}_i'\boldsymbol{\beta})>1-\epsilon_i} \hat{\lambda} y_i h(\mathbf{x}_0)'h(\mathbf{x}_i)$. The best kernel function is also selected through cross validation. This project tested linear, polynomial, sigmoidal, and radial kernels.

The last issue to address with SVMs is the multi-class case. The common approach (and the one used in this project) is "one-versus-all" (OVA). Essentially, the SVM is fit on each class, combining the other classes into a single "other" category each time. The classifier that has the most distance between the given observation $\mathbf{x}_i$ and the hyperplane is the predicted class. For more details on OVA, see Rifkin and Klautau (2004).

4 Results The classification methods described in Section 3 were applied under two different contexts. One is to identify the speaker of single quotes. The quotes range in length from 31 to

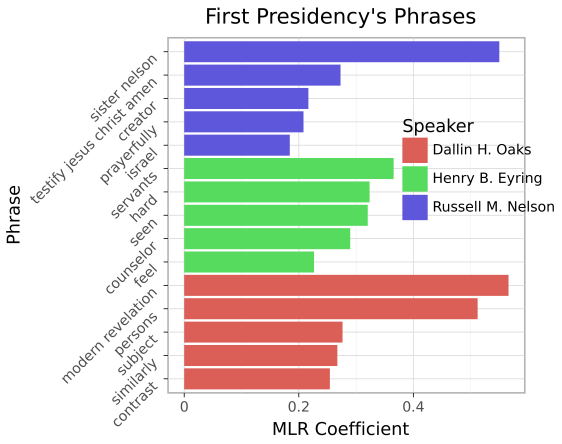

Figure 5: Phrases that best predict the First Presidency in MLR.

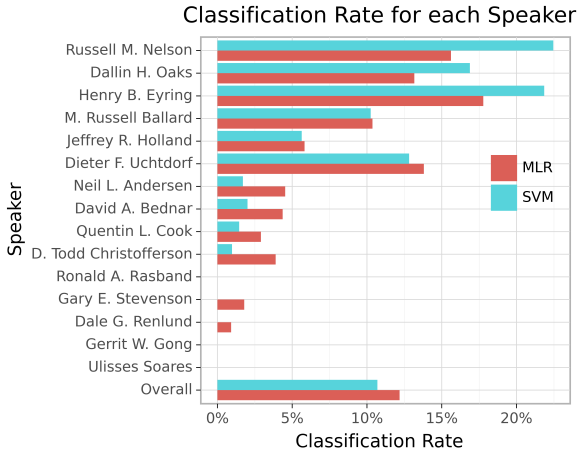

Figure 6: Prediction accuracy by speaker for MLR and SVM.

2,318 characters. The other is predicting if a quote is from 2020 or not, using only the talks given by President Dallin H. Oaks. For all predictions, group imbalance was a significant issue. Of all the quotes, 44% came from just the First Presidency (three speakers). The 3 youngest members, Elders Dale G. Renlund, Gerrit W. Gong, and Ulisses Soares, had only 4% of the quotes. Additionally, only 4% of the quotes from Dallin H. Oaks come from 2020. All reported prediction metrics are performed on the out-of-sample predictions.

For the author attribution predictions, the best lasso regularization coefficient for MLR turned out to be 10 (found through cross-validated accuracy). The regularization left, on average, 35 predictor phrases for each speaker (61 on average for First-Presidency members and 14 for youngest Quorum members). Figure 5 shows the 5 phrases with the highest regression coefficients in predicting the First-Presidency members. The F-score is a measurement of prediction accuracy. It is the harmonic mean of precision and recall for each class, combined together as a weighted mean, with weights based on how many observations belong to each class. The closer this value is to 1, the better. The F-score of the MLR fit is .107.

The support vector machine predicted speakers best with a margin error parameter λ of 1 and a sigmoid kernel function. The F-score for the SVM in predicting the text authors was also .107. One interesting trend is that for speakers with more text (the first presidency and the older members of the Quorum of the 12), SVMs were superior to MLR. Figure 6 shows the classification rate (recall) of each speaker by each of the two classification methods. It is surprising how similar these two methods were to each other in prediction, because many researchers hypothesize that SVM is one of the best methods and few recommend MLR.

These two methods were also applied on quotes from Dallin H. Oaks. Again, group imbalance does make prediction difficult. The prediction accuracy here can be measured by F-score and also area-under-the-curve (AUC). AUC is the total area under the ROC curve. The ROC shows the precision-recall tradeoff at various cutoff for which predicted probabilities are classified as "before 2020" or "after 2020". If prediction is perfect, then the AUC is 1, and the worst that it can be is .5 (random prediction). The AUC is .70 for MLR classification and .84 for SVM, so prediction is mediocre. The F-score is .95 and .94, for MLR and SVM, respectively. The real problem is that SVM predicts that none of the quotes are from 2020. MLR predicted only 3 quotes from 2020 (out of 38 total out-of-sample 2020 quotes). Even though the F-scores and AUC are not terrible, recall for the 2020 quotes is less than 8% for both methods.

5 Conclusion Classifying speakers or classifying a specific year of a given speaker with a single quote is a difficult problem. This project uses TF-IDF scores on n-grams of talks and speeches and

two classification methods to perform NLP author attribution. Speakers with more observations have better prediction accuracy in both methods, but SVM was superior for those with more. Neither method was particularly good at prediction, with the highest recall for any speaker of .25. Predicting President Oaks' 2020 talks was also problematic. Both methods gave 2020 quote recall $< .08$. While prediction is not particularly easy in this problem, Many interesting discoveries were made, such as the most important words for predicting the first presidency members, or the changes in word usage over time. More exploration could reveal other interesting trends.

One way to improve this methodology is to replace the n-grams used in this project with the syntactic n-grams proposed in Sidorov et al. (2014). In this paper, these improved features resulted in better author attribution when using SVM. Another weakness of this project is the class imbalance in the quotes. One way to combat this problem may have been to instead of taking the 2.500 phrases with largest TF-IDF scores, take the phrases with highest TF-IDF scores *for each speaker*. In taking the overall mean, the current method is potentially removing words that might be very useful in predicting the speakers who have fewer quotes. Many researchers believe that SVM, naive bayes, and deep learning methodology are the best classification techniques in this NLP setting. One expansion to this project could be to use naive bayes and deep learning and compare to SVM.

References

Collins, J. (2003), 'Variations in written english: Characterizing the authors' rhetorical language choices across corpora of published texts', *Diss. Carnegi Mellon University* .

Eder, M. (2011), 'Style-markers in authorship attribution: a cross-language study of the authorial fingerprint', *Studies in Polish Linguistics* **6**(1).

Kruczek, J., Kruczek, P. and Kuta, M. (2020), Are n-gram categories helpful in text classification?, *in* 'International Conference on Computational Science', Springer, pp. 524–537.

Rifkin, R. and Klautau, A. (2004), 'In defense of one-vs-all classification', *Journal of machine learning research* **5**(Jan), 101–141.

Shi, C., Xu, C. and Yang, X. (2009), 'Study of tfidf algorithm', *Journal of Computer Applications* **29**(6), 167–170.

Sidorov, G., Velasquez, F., Stamatatos, E., Gelbukh, A. and Chanona-Hernández, L. (2014), 'Syntactic n-grams as machine learning features for natural language processing', *Expert Systems with Applications* **41**(3), 853–860.

Stack, P. F., Davidson, L. and Noyce, D. (2020), 'Latter-day saint leaders tackle big issues: elections, racism, protests and pandemic', *The Salt Lake Tribune* .
URL: *https://www.sltrib.com/religion/2020/10/03/lds-general-conference/*

A Python Code

```
import sys
import requests
import numpy as np
import pandas as pd
from sklearn.svm import SVC
from bs4 import BeautifulSoup
from datetime import datetime
from sklearn.metrics import f1_score
from sklearn.metrics import confusion_matrix
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer


##############################################################################
#           Scraping General Conference Talks

church_URL = 'https://www.churchofjesuschrist.org/'

escapes = ''.join([chr(char) for char in range(1, 32)])
translator = str.maketrans('', '', escapes)

all_speakers_URL = 'general-conference/speakers?lang=eng'

all_speakers_page = requests.get(church_URL + all_speakers_URL)
all_speaker_soup = BeautifulSoup(all_speakers_page.content, 'lxml')

all_speaker_links_soup = all_speaker_soup.find_all('div', "lumen-tile__title")
all_links = [spkr.find('a')['href'][1:] for spkr in all_speaker_links_soup]
```

```
links_to_use = all_links[:15]

page_num = str(1)

talks = pd.DataFrame(columns = ['Speaker', 'Date', 'Title', 'URL'])

for speaker_link in links_to_use:
    page = requests.get(church_URL + speaker_link)

    scripture_soup = BeautifulSoup(page.content, 'lxml')
    all_links = scripture_soup.find_all('a', 'pages-nav__list-item__anchor')
    if len(all_links) > 0:
        link_texts = [int(link.text) if len(link.text) == 1 else 0
                      for link in all_links]
        num_pages = np.max(link_texts)
    else:
        num_pages = 1

    for this_page_num in range(1,num_pages+1):
        this_URL = church_URL + speaker_link + "&page=" + str(this_page_num)
        this_page = requests.get(this_URL)
        speaker_soup = BeautifulSoup(this_page.content, 'lxml')

        this_speaker = speaker_soup.find('h1').get_text()

        talk_titles_soup = speaker_soup.find_all('div', "lumen-tile__title")
        these_titles = [title.get_text().translate(translator)
                        for title in talk_titles_soup]

        talk_dates_soup = speaker_soup.find_all('div', "lumen-tile__content")
        these_dates_str = [date.get_text().translate(translator)
                           for date in talk_dates_soup]
        these_dates = [datetime.strptime(date_str, "%B %Y")
                       for date_str in these_dates_str]

        talk_links_soup = speaker_soup.find_all('a', "lumen-tile__link")
        these_URLs = [church_URL + link['href'][1:]
                      for link in talk_links_soup]

        these_talks = pd.DataFrame({
                        'Speaker':[this_speaker for i in these_URLs],
                        'Date':these_dates,
                        'Title':these_titles,
                        'URL':these_URLs})

        talks = talks.append(these_talks, ignore_index = True)

all_content = []

for talk_row in range(np.shape(talks)[0]):
    perc_comp = talk_row / np.shape(talks)[0] * 100
    sys.stdout.write(f"\rWebscrape Progress: {int(np.round(perc_comp))}%")
    talk_URL = talks['URL'][talk_row]
    talk_page = requests.get(talk_URL)
    talk_soup = BeautifulSoup(talk_page.content, 'lxml')
    ids = [p.get('id') for p in talk_soup.find_all('p')]
    paragraphs = [idx[0] == 'p' and idx[1].isdigit() if idx != None else False
                  for idx in ids]
    paragraph_ids = [ids[paragraph_index]
                     for paragraph_index in list(np.where(paragraphs)[0])]

    content = [talk_soup.find(id = this_id).get_text().translate(translator)
               for this_id in paragraph_ids]
    all_content += [content]
```

```
    sys.stdout.flush()
sys.stdout.write("\rWebscrape Progress: 100%\n")
sys.stdout.write("Webscrape complete!\n")

quotes = pd.DataFrame(columns = list(talks.columns))

for i in range(np.shape(talks)[0]):
    perc_comp = i / np.shape(talks)[0] * 100
    sys.stdout.write(f"\rData Compiling Progress: {int(np.round(perc_comp))}%")
    num_quotes = len(all_content[i])
    talk_speaker = [talks.iloc[i,0] for x in range(num_quotes)]
    talk_date = [talks.iloc[i,1] for x in range(num_quotes)]
    talk_title = [talks.iloc[i,2] for x in range(num_quotes)]
    talk_URL = [talks.iloc[i,3] for x in range(num_quotes)]

    talk_quote_info = pd.DataFrame({'Speaker':talk_speaker,
                                    'Date':talk_date,
                                    'Title':talk_title,
                                    'URL':talk_URL,
                                    'Content':all_content[i]})
    quotes = quotes.append(talk_quote_info, ignore_index = True)
sys.stdout.write("\rData Compiling Progress: 100%\n")
sys.stdout.write("Compilation complete!\n")

quotes.to_csv('Talk_Quotes_Data.csv', index = False)


##############################################################################
#           Scraping BYU Speeches

speeches_URL = 'https://speeches.byu.edu'

speakers = list(pd.read_csv('Talk_Quotes_Data.csv')['Speaker'].unique())
speakers_format = [spkr.lower().replace(' ','-').replace('.','')
                   for spkr in speakers]

all_speaker_links = [speeches_URL + '/speakers/' + spkr + '/'
                     for spkr in speakers_format]

speeches = pd.DataFrame(columns = ['Speaker', 'Date', 'Title', 'URL'])

escapes = ''.join([chr(char) for char in range(1, 32)])
translator = str.maketrans('', '', escapes)

for spkr_link in all_speaker_links:
    spkr_page = requests.get(spkr_link)
    spkr_soup = BeautifulSoup(spkr_page.content, 'lxml')
    all_talks = spkr_soup.find_all('article', 'card-card--reduced')
    this_speaker = spkr_soup.find('h1', 'speaker-listing--name').text
    for talk in all_talks:
        this_Date = datetime.strptime(talk
                    .find('div', 'card--bylines-card--bylines--reduced')
                    .text
                    .translate(translator), "%B %d, %Y")
        this_URL = talk.find('a')['href']
        this_Title = talk.find('h2').text.translate(translator)
        this_speech = pd.DataFrame({'Speaker':[this_speaker],
                                    'Date':[this_Date],
                                    'Title':[this_Title],
                                    'URL':[this_URL]})
        speeches = speeches.append(this_speech, ignore_index = True)


all_content = []
```

```
unavail_text = 'The text for this speech is unavailable.'
int_prop = 'Intellectual Reserve'
rights = 'All rights reserved.'

rows_to_rm = []

for speech_row in range(np.shape(speeches)[0]):
    perc_comp = speech_row / np.shape(speeches)[0] * 100
    sys.stdout.write(f"\rWebscrape Progress: {int(np.round(perc_comp))}%")
    speech_URL = speeches['URL'][speech_row]
    speech_page = requests.get(speech_URL)
    speech_soup = BeautifulSoup(speech_page.content, 'lxml')
    all_paragraphs = speech_soup.findChildren('p',
                                              recursive = True,
                                              attrs={'class': None})
    all_text = [p.text.translate(translator) for p in all_paragraphs]
    if len(all_text) > 2:
        end_idx = np.max(np.where([' amen.' in text
                                   or ' Amen.' in text
                                   or int_prop in text
                                   or unavail_text in text
                                   or rights in text
                                   for text in all_text]))
        if (unavail_text in all_text[end_idx] or
            int_prop in all_text[end_idx] or
            rights in all_text[end_idx]):
            end_idx += -1
            all_text = [text
                        for text in all_text
                        if 'Speech highlights' not in text]
        content = all_text[:(end_idx+1)]
        all_content += [content]
    else:
        rows_to_rm += [speech_row]
    sys.stdout.flush()

speeches = speeches[~speeches.index.isin(rows_to_rm)]

sys.stdout.write("\rWebscrape Progress: 100%\n")
sys.stdout.write("Webscrape complete!\n")

quotes = pd.DataFrame(columns = list(speeches.columns))

for i in range(np.shape(speeches)[0]):
    perc_comp = i / np.shape(speeches)[0] * 100
    sys.stdout.write(f"\rData Compiling Progress: {int(np.round(perc_comp))}%")
    num_quotes = len(all_content[i])
    speech_speaker = [speeches.iloc[i,0] for x in range(num_quotes)]
    speech_date = [speeches.iloc[i,1] for x in range(num_quotes)]
    speech_title = [speeches.iloc[i,2] for x in range(num_quotes)]
    speech_URL = [speeches.iloc[i,3] for x in range(num_quotes)]

    speech_quote_info = pd.DataFrame({'Speaker':speech_speaker,
                                      'Date':speech_date,
                                      'Title':speech_title,
                                      'URL':speech_URL,
                                      'Content':all_content[i]})
    quotes = quotes.append(speech_quote_info, ignore_index = True)
sys.stdout.write("\rData Compiling Progress: 100%\n")
sys.stdout.write("Compilation complete!\n")

quotes.to_csv('Speech_Quotes_Data.csv', index = False)
```

```
##############################################################################
#           Feature Selection

talks = pd.read_csv('Talk_Quotes_Data.csv', parse_dates = [1])
speeches = pd.read_csv('Speech_Quotes_Data.csv', parse_dates = [1])

all_talks = talks.append(speeches, ignore_index=True)
all_talks = all_talks[all_talks.Content.str.len() > 30]
np.shape(all_talks)

del talks, speeches

X_train_all, X_test_all, y_train, y_test = train_test_split(all_talks,
                                                            all_talks.Speaker,
                                                            test_size=0.2)
X_train = X_train_all.Content
X_test = X_test_all.Content
speaker_dict = {k:v for v, k in enumerate(all_talks.Speaker.unique())}
y_train = [speaker_dict[speaker] for speaker in list(y_train)]
y_test = [speaker_dict[speaker] for speaker in list(y_test)]
to_speaker_dict = {v:k for v, k in enumerate(all_talks.Speaker.unique())}


vectorizer = TfidfVectorizer(analyzer = 'word',
                             stop_words = 'english',
                             max_features = 2500,
                             ngram_range = (1,5))
vectorizer.fit(list(all_talks.Content))
tfidf_X_train = vectorizer.transform(X_train)
tfidf_X_test = vectorizer.transform(X_test)


scaler = StandardScaler(with_mean=False)
tfidf_X_train = scaler.fit_transform(tfidf_X_train)
tfidf_X_test = scaler.transform(tfidf_X_test)
feature_names = vectorizer.get_feature_names()



##############################################################################
#           Modeling Speaker Classification


### MLR
# Grid search for MLR regularization
MLR_param_grid = {'C' : np.logspace(-4, 1, 6)}

# MLR model details
MLR_mod = LogisticRegression(random_state=0,
                             multi_class = 'multinomial',
                             penalty = 'l1',
                             solver = 'saga')

# MLR fitting
MLR_cv=GridSearchCV(MLR_mod, MLR_param_grid, cv=4 , verbose = 5, n_jobs=-1)
MLR_cv.fit(tfidf_X_train, y_train)

### SVM
# Grid search for SVM regularization and kernel function
SVM_param_grid = {'C' : np.logspace(-4, 2, 4),
              'kernel' : ['linear', 'poly', 'rbf', 'sigmoid']}

# SVM model details
```

```
SVM_model = SVC(verbose = True)

# SVM fitting
SVM_cv = GridSearchCV(SVM_model,
                      SVM_param_grid,
                      cv = 2, verbose = 8, n_jobs = -1)
SVM_cv.fit(tfidf_X_train, y_train)

##############################################################################
#           Prediction Measurements for Speaker Classification

### MLR
# MLR Best Regularization: lambda = 1/C
1/MLR_cv.best_params_['C']
# MLR Run time
MLR_cv.refit_time_

# MLR Prediction
out_predictions_MLR = MLR_cv.predict(tfidf_X_test)

# MLR Speaker-Specific Recall
MLR_Results = pd.DataFrame({'Preds':out_predictions_MLR, 'Act':y_test})
MLR_Results['Accuracy'] = MLR_Results.Preds == MLR_Results.Act
MLR_Results.groupby('Act')['Accuracy'].mean()

# MLR F-score
f1_score(y_test, out_predictions_MLR, average = "weighted")

# MLR Confusion Matrix
pd.DataFrame(confusion_matrix(y_test, out_predictions_MLR),
             index = speakers, columns = speakers)

### SVM
# SVM Best Regularization and Kernel
SVM_cv.best_params_
# SVM Run time
SVM_cv.refit_time_

# SVM Prediction
out_predictions_SVM = SVM_cv.predict(tfidf_X_test)

# SVM Speaker-Specific Recall
SVM_Results = pd.DataFrame({'Preds':out_predictions_SVM, 'Act':y_test})
SVM_Results['Accuracy'] = SVM_Results.Preds == SVM_Results.Act
SVM_Results.groupby('Act')['Accuracy'].mean()

# SVM F-score
f1_score(y_test, out_predictions_SVM, average = "weighted")

# SVM Confusion Matrix
pd.DataFrame(confusion_matrix(y_test, out_predictions_SVM),
             index = speakers, columns = speakers)


##############################################################################
#           Modeling Dallin H. Oaks 2020

# Data prep for Dallin H. Oaks
tfidf_X_train_oaks = tfidf_X_train[X_train_all.Speaker=="Dallin H. Oaks"]
tfidf_X_test_oaks = tfidf_X_test[X_test_all.Speaker=="Dallin H. Oaks"]

y_train_oaks_full = pd.concat([X_train_all.Speaker=="Dallin H. Oaks",
                               X_train_all.Date.astype(str).str.startswith("2020")],
                              axis = 1)
```

```
y_train_oaks = y_train_oaks_full[y_train_oaks_full.Speaker].Date
y_test_oaks_full = pd.concat([X_test_all.Speaker=="Dallin H. Oaks",
                              X_test_all.Date.astype(str).str.startswith("2020")],
                              axis = 1)
y_test_oaks = y_test_oaks_full[y_test_oaks_full.Speaker].Date


### MLR
# Grid search for MLR regularization
MLR_param_grid_oaks = {'C' : np.logspace(-10, 10, 20)}

# MLR model details
MLR_mod_oaks = LogisticRegression(random_state=0,
                                  penalty = 'l1',
                                  solver = 'saga')

# MLR fitting
MLR_cv_oaks = GridSearchCV(MLR_mod,
                           MLR_param_grid_oaks,
                           cv=4 , verbose = 5, n_jobs=-1)
MLR_cv_oaks.fit(tfidf_X_train_oaks, y_train_oaks)

### SVM
# Grid search for SVM regularization and kernel function
SVM_param_grid_oaks = {'C' : np.logspace(-4, 4, 8),
              'kernel' : ['linear', 'poly', 'rbf', 'sigmoid']}

# SVM model details
SVM_model_oaks = SVC(verbose = True)

# SVM fitting
SVM_cv_oaks = GridSearchCV(SVM_model_oaks,
                      SVM_param_grid_oaks,
                      cv=2, verbose = 8, n_jobs = -1)
SVM_cv_oaks.fit(tfidf_X_train_oaks, y_train_oaks)

##############################################################################
#            Prediction Measurements for Dallin H. Oaks 2020

### MLR
# MLR Best Regularization: lambda = 1/C
1/MLR_cv_oaks.best_params_['C']
# MLR Run time
MLR_cv_oaks.refit_time_

# MLR Prediction
out_predictions_MLR_oaks = MLR_cv_oaks.predict(tfidf_X_test_oaks)

# MLR Speaker-Specific Recall
MLR_Results_oaks = pd.DataFrame({'Preds':out_predictions_MLR_oaks*1,
                                 'Act':y_test_oaks*1})
MLR_Results_oaks['Accuracy'] = MLR_Results_oaks.Preds == MLR_Results_oaks.Act
MLR_Results_oaks.groupby('Act')['Accuracy'].mean()

# MLR F-score
f1_score(y_test_oaks, out_predictions_MLR_oaks, average = "weighted")

# MLR Confusion Matrix
pd.DataFrame(confusion_matrix(y_test_oaks, out_predictions_MLR_oaks),
             index = ['Before', 'After'], columns = ['Before', 'After'])

### SVM
# SVM Best Regularization and Kernel
SVM_cv_oaks.best_params_
```

```
# SVM Run time
SVM_cv_oaks.refit_time_

# SVM Prediction
out_predictions_SVM_oaks = SVM_cv_oaks.predict(tfidf_X_test_oaks)

# SVM Speaker-Specific Recall
SVM_Results_oaks = pd.DataFrame({'Preds':out_predictions_SVM_oaks*1,
                                 'Act':y_test_oaks*1})
SVM_Results_oaks['Accuracy'] = SVM_Results_oaks.Preds == SVM_Results_oaks.Act
SVM_Results_oaks.groupby('Act')['Accuracy'].mean()

# SVM F-score
f1_score(y_test_oaks, out_predictions_SVM_oaks, average = "weighted")

# SVM Confusion Matrix
pd.DataFrame(confusion_matrix(y_test_oaks, out_predictions_SVM_oaks),
             index = ['Before', 'After'], columns = ['Before', 'After'])


##############################################################################
#           Feature Selection for Talk-level

all_talks.Content += "␣"
talk_level = (all_talks
                  .groupby(['Speaker','Date'])['Content']
                  .sum()
                  .reset_index()
                  .Content)

(X_train_all_talks,
     X_test_all_talks,
     y_train_talks,
     y_test_talks) = train_test_split(all_talks,
                                      all_talks.Speaker,
                                      test_size=0.2)
X_train_talks = X_train_all_talks.Content
X_test_talks = X_test_all_talks.Content
y_train_talks = [speaker_dict[speaker] for speaker in list(y_train_talks)]
y_test_talks = [speaker_dict[speaker] for speaker in list(y_test_talks)]
to_speaker_dict = {v:k for v, k in enumerate(all_talks.Speaker.unique())}

tfidf_X_train_talks = vectorizer.transform(X_train_talks)
tfidf_X_test_talks = vectorizer.transform(X_test_talks)

tfidf_X_train_talks = scaler.fit_transform(tfidf_X_train_talks)
tfidf_X_test_talks = scaler.transform(tfidf_X_test_talks)

##############################################################################
#           Modeling Talk-Level

### MLR
# Grid search for MLR regularization
MLR_param_grid_talks = {'C' : np.logspace(-10, 10, 20)}

# MLR model details
MLR_mod_talks = LogisticRegression(random_state=0,
                                   penalty = 'l1',
                                   solver = 'saga')

# MLR fitting
MLR_cv_talks = GridSearchCV(MLR_mod_talks,
                            MLR_param_grid_talks,
                            cv=4 , verbose = 5, n_jobs=-1)
```

```
MLR_cv_talks.fit(tfidf_X_train_talks, y_train_talks)

### SVM
# Grid search for SVM regularization and kernel function
SVM_param_grid_talks = {'C' : np.logspace(-4, 4, 8),
              'kernel' : ['linear', 'poly', 'rbf', 'sigmoid']}

# SVM model details
SVM_model_talks = SVC(verbose = True)

# SVM fitting
SVM_cv_talks = GridSearchCV(SVM_model_talks,
                        SVM_param_grid_talks,
                        cv=2, verbose = 8, n_jobs = -1)
SVM_cv_talks.fit(tfidf_X_train_talks, y_train_talks)

################################################################################
#            Prediction Measurements for Talk-Level

### MLR
# MLR Best Regularization: lambda = 1/C
1/MLR_cv_talks.best_params_['C']
# MLR Run time
MLR_cv_talks.refit_time_

# MLR Prediction
out_predictions_MLR_talks = MLR_cv_talks.predict(tfidf_X_test_talks)

# MLR Speaker-Specific Recall
MLR_Results_talks = pd.DataFrame({'Preds':out_predictions_MLR_talks,
                                  'Act':y_test_talks})
MLR_Results_talks['Accuracy'] = MLR_Results_talks.Preds==MLR_Results_talks.Act
MLR_Results_talks.groupby('Act')['Accuracy'].mean()

# MLR F-score
f1_score(y_test_talks, out_predictions_MLR_talks, average = "weighted")

# MLR Confusion Matrix
pd.DataFrame(confusion_matrix(y_test_talks, out_predictions_MLR_talks),
             index = speakers, columns = speakers)

### SVM
# SVM Best Regularization and Kernel
SVM_cv_talks.best_params_
# SVM Run time
SVM_cv_talks.refit_time_

# SVM Prediction
out_predictions_SVM_talks = SVM_cv_talks.predict(tfidf_X_test_talks)

# SVM Speaker-Specific Recall
SVM_Results_talks = pd.DataFrame({'Preds':out_predictions_SVM_talks,
                                  'Act':y_test_talks})
SVM_Results_talks['Accuracy'] = SVM_Results_talks.Preds==SVM_Results_talks.Act
SVM_Results_talks.groupby('Act')['Accuracy'].mean()

# SVM F-score
f1_score(y_test_talks, out_predictions_SVM_talks, average = "weighted")

# SVM Confusion Matrix
pd.DataFrame(confusion_matrix(y_test_talks, out_predictions_SVM_talks),
             index = speakers, columns = speakers)
```