Chris Richardson Oct 3, 2022 ADS-509-Fall Github Link: https://github.com/CFRichardson/USD_ADS_509_HW4 Naive Bayes on Political Text In this notebook we use Naive Bayes to explore and classify political data. See the README.md for full details. **Notebook Setup** In [1]: import nltk import numpy as np import random import sqlite3 from collections import Counter, defaultdict In [2]: def conv features(text, fw, include false=False) : """Given some text, this returns a dictionary holding the feature words. * text: a piece of text in a continuous string. Assumes text has been cleaned and case folded. * fw: the *feature words* that we're considering. A word in `text` must be in fw in order to be returned. This prevents us from considering very rarely occurring words. Returns: A dictionary with the words in `text` that appear in `fw`. Words are only counted once. If `text` were "quick quick brown fox" and `fw` = {'quick', 'fox', 'jumps'}, then this would return a dictionary of {'quick' : True, 'fox': True } ret dict = dict() present tokens = set(text.split()) for token in feature words: if token in present tokens: ret dict[token] = True else: if include false: # include false ret dict[token] = False return(ret dict) # added libraries In [3]: import os import pandas as pd import scipy import string import re from nltk.corpus import stopwords from tqdm import tqdm # --- functions from past homework assignments for this course --# Some punctuation variations punctuation = set(string.punctuation) # speeds up comparison # somehow, to_add[1] != to add[2] to add = ['`',''',''','•','>','«','×'] punctuation.update(to_add) punctuation.remove('#') # Stopwords sw = stopwords.words("english") def contains_emoji(s): emoji_count = emoji.emoji_count(s) return(emoji_count > 0) def prepare(text, pipeline) : Chandler, John August 22, 2022 ADS 509 Module 3: Group Comparison Code Version: Git commit 0405f0f35f67edf62f95bba5052cc11efbda26c9 NLP Pipeline Transformer https://github.com/37chandler/ads-tm-group-comp/blob/main/Group%20Comparison.ipynb tokens = str(text)for transform in pipeline : tokens = transform(tokens) return (tokens) def remove_punctuation(text, punct_set=punctuation): Chandler, John August 22, 2022 ADS 509 Module 3: Group Comparison Code Version: Git commit 0405f0f35f67edf62f95bba5052cc11efbda26c9 NLP Punctuation Remover https://github.com/37chandler/ads-tm-group-comp/blob/main/Group%20Comparison.ipynb return("".join([ch for ch in text if ch not in punct_set])) def remove_stop(text) : tokens = text.split() tokens = [token for token in tokens if token not in sw] string_ = ' '.join(tokens) return(string) def tokenize(text) : """ Splitting on whitespace rather than the book's tokenize function. That function will drop tokens like '#hashtag' or '2A', which we need for Twitter. """ tokens = text.split() return (tokens) Part 1: Exploratory Naive Bayes We'll first build a NB model on the convention data itself, as a way to understand what words distinguish between the two parties. This is analogous to what we did in the "Comparing Groups" class work. First, pull in the text for each party and prepare it for use in Naive Bayes. In [4]: convention_db = sqlite3.connect("2020 Conventions.db") convention cur = convention db.cursor() In [5]: convention_data = [] query = ''' SELECT text, party FROM conventions 1.1.1 query results = convention cur.execute(query) text prep pipeline = [str.lower, remove punctuation, remove stop] for row in query results : text = prepare(text=row[0], pipeline=text_prep_pipeline) party = row[1]convention data.append([text,party]) Let's look at some random entries and see if they look right. In [6]: random.choices(convention data, k=2) [['faced president cowardice joe biden man proven courage restore moral compass confronting challenges hiding u Out[6]: ndermining elections keep job', 'Democratic'], ['washington dc', 'Democratic']] If that looks good, we now need to make our function to turn these into features. In my solution, I wanted to keep the number of features reasonable, so I only used words that occur at least word_cutoff times. Here's the code to test that if you want it. In [7]: word_cutoff = 5 tokens = [w for t, p in convention_data for w in t.split()] word dist = nltk.FreqDist(tokens) feature_words = set() for word, count in word_dist.items() : if count > word_cutoff : feature_words.add(word) print(f"With a word cutoff of {word_cutoff}, we have {len(feature_words)} as features in the model.") With a word cutoff of 5, we have 2383 as features in the model. In [8]: assert(len(feature words)>0) assert(conv features("donald is the president", feature words) == {'donald':True,'president':True}) assert(conv features("people are american in america", feature words) === { 'america':True, 'american':True, "people":True}) Now we'll build our feature set. Out of curiosity I did a train/test split to see how accurate the classifier was, but we don't strictly need to since this analysis is exploratory. Classifier 1 W/Out False Values In [9]: featuresets = [(conv_features(text, feature_words), party) for (text, party) in tqdm(convention_data)] 2541/2541 [00:00<00:00, 6732.32it/s] In [10]: random.seed(20220507) random.shuffle(featuresets) test size = 500 In [11]: test_set, train_set = featuresets[:test_size], featuresets[test_size:] classifier = nltk.NaiveBayesClassifier.train(train_set) print(nltk.classify.accuracy(classifier, test_set)) 0.498 In [58]: classifier.show_most_informative_features(5) Most Informative Features china = True Republ : Democr = 25.8 : 1.0 votes = True Democr : Republ = 23.8 : 1.0 enforcement = True Republ : Democr = 21.5 : 1.0 destroy = True Republ : Democr = 19.2 : 1.0 Classifier 2 W/ False Values feature_sets = [(conv_features(text, feature words, include false=True), party) for (text, party) in tqdm(convention data)] random.seed(20220507) random.shuffle(featuresets) test set, train set = feature sets[:test size], feature sets[test size:] classifier wFalse = nltk.NaiveBayesClassifier.train(train set) print(nltk.classify.accuracy(classifier wFalse, test set)) | 2541/2541 [00:00<00:00, 3272.30it/s] 0.772 classifier_wFalse.show_most_informative_features(5) In [14]: Most Informative Features Republ : Democr =
Democr : Republ =
Republ : Democr =
Republ : Democr = radical = True 35.1 : 1.0 votes = True 35.0 : 1.0 enforcement = True 18.1 : 1.0 freedoms = True
 signed = True 16.5 : 1.0 Republ : Democr = 16.5 : 1.0 My Observations Strangely enough, even though we include false values which increases test accuracy by roughly 20%, most informative features is the exact same for both classifiers. Only difference is False values in our classifier 1 are None instead of False as shown in classifier 2. Another interesting find is that only 2 features, votes and climate, are the only two Democratic Party dominant words out of the top 50 informative features. Thus, it appears that the classifier favors in recognizing if a text is Republican or not Republican. Part 2: Classifying Congressional Tweets In this part we apply the classifer we just built to a set of tweets by people running for congress in 2018. These tweets are stored in the database congressional_data.db . That DB is funky, so I'll give you the query I used to pull out the tweets. Note that this DB has some big tables and is unindexed, so the query takes a minute or two to run on my machine. **Data Pull** In [15]: cong_db = sqlite3.connect("congressional_data.db") cong_cur = cong_db.cursor() In [16]: %%time query = ''' SELECT DISTINCT cd.candidate, cd.party, tw.tweet text FROM candidate data cd INNER JOIN tweets tw ON cd.twitter handle = tw.handle AND cd.candidate == tw.candidate AND cd.district == tw.district WHERE cd.party in ('Republican', 'Democratic') AND tw.tweet text NOT LIKE '%RT%' results = cong db.execute(query) results = list(results) # Just to store it, since the query is time consuming CPU times: user 4.61 s, sys: 1.69 s, total: 6.31 s Wall time: 15.4 s In [17]: # Regex Pattern from by stackoverflow user "zx81" on # https://stackoverflow.com/questions/24399820/expression-to-remove-url-links-from-twitter-tweet def html remover(str): pattern = r'(http)\S+' text = re.sub(pattern, '', str_) return(text) **Text Prep** In [18]: tweet_data = [] for row in tqdm(results): text_prep_pipeline = [str.lower, html remover, remove punctuation, remove stop] text = prepare(text=row[2].decode('utf-8'), pipeline=text prep pipeline) party = row[1]tweet_data.append([text, party]) | 664656/664656 [00:26<00:00, 25532.27it/s] There are a lot of tweets here. Let's take a random sample and see how our classifer does. I'm guessing it won't be too great given the performance on the convention speeches... In [56]: random.seed(20201014) tweet_data_sample = random.sample(tweet_data, k=2) Classification In [57]: for tweet, party in tweet data sample: estimated party = classifier wFalse.classify(conv features(tweet, feature_words, False)) print(f"Clean tweet: {tweet}\n\n") print(f"Actual party: {party}\n", f'Classifier prediction {estimated party}') print("--"*10) Clean tweet: catch realerincruz show right Actual party: Republican Classifier prediction Republican _____ Clean tweet: thank pennsylvania governor tom wolf register vote pennsylvania online Actual party: Democratic Classifier prediction Democratic Now that we've looked at it some, let's score a bunch and see how we're doing. Predictions w/ False Values In [22]: # dictionary of counts by actual party and estimated party. # first key is actual, second is estimated parties = ['Republican','Democratic'] results = defaultdict(lambda: defaultdict(int)) for p in parties : for p1 in parties : results[p][p1] = 0num to score = 10000 random.shuffle(tweet data) for idx, tp in enumerate(tweet_data) : tweet, party = tp # get the estimated party estimated party = classifier_wFalse.classify(conv_features(tweet, feature words, results[party][estimated_party] += 1 if idx > num_to_score : break results defaultdict(<function __main__.<lambda>()>, {'Republican': defaultdict(int, {'Republican': 57, 'Democratic': 4180}), 'Democratic': defaultdict(int, {'Republican': 45, 'Democratic': 5720})}) Predictions w/out False Values In [23]: # dictionary of counts by actual party and estimated party. # first key is actual, second is estimated parties = ['Republican','Democratic'] results = defaultdict(lambda: defaultdict(int)) for p in parties : for pl in parties : results[p][p1] = 0num to score = 10000random.shuffle(tweet_data) for idx, tp in enumerate(tweet data) : tweet, party = tp # get the estimated party estimated party = classifier wFalse.classify(conv features(tweet, feature words, False)) results[party][estimated party] += 1 if idx > num to score : break results defaultdict(<function main .<lambda>()>, {'Republican': defaultdict(int, {'Republican': 3785, 'Democratic': 557}), 'Democratic': defaultdict(int, {'Republican': 4897, 'Democratic': 763})}) Reflections Write a little about what you see in the results_ It appears that our classifier favors classifying documents that are written by Democratics. As we see with Democratic documents, the classifier without False Values overfitted and practically thought every document is from the Democratic Party; with a recall of 1.35% (=57/(57 + 4180)) when attempting to classify a Republican Document and 99.22% recall score when classifying a Democratic Document. While the classifier without False values overfitted again, but this time the classifier practically always classifying a document as Republican with an recall of 88.17% (=3785/(3785 + 557)) for Republican Documents but a measly recall of 13.48% when classifying for Democratic Documents. Clearly the utilization of sparse information leads to overfitting, most likely due to the curse of dimensionality (too many features). Recall = TP/(TP + FN)References zx81 (2014, June 25). Expression to remove URL links from Twitter tweet. StackOverFlow. https://stackoverflow.com/questions/24399820/expression-to-remove-url-links-from-twitter-tweet