Module 4-Political Naive Bayes

February 5, 2024

0.1 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.

[ ]:
```
import os
import re
import sqlite3
import random
import numpy as np
from collections import Counter, defaultdict
from string import punctuation

np.int = np.int_
np.float = np.float_
import nltk
import shutil
```

[ ]:
```
#first, download the stopwords.zip file from https://www.nltk.org/nltk_data/␣
 ↪and then extract it
source_dir = '/Users/calebmccurdy/downloads/stopwords'
dest_dir = '/Users/calebmccurdy/nltk_data/corpora/stopwords'

if os.path.exists(dest_dir):
    shutil.rmtree(dest_dir)
shutil.move(source_dir, dest_dir)

from nltk.corpus import stopwords
sw = stopwords.words("english")
```

[ ]:
```
punctuation = set(punctuation) # speeds up comparison

# Albrecht, J., Ramachandran, S., & Winkler, C. (2020). Blueprints for text␣
 ↪analytics using Python. O'Reilly.
RE_TOKEN = re.compile(r"""
                ( [#]?[@\w'’\.\-\:]*\w     # words, hash tags and email adresses
                | [:;<]\-?[\)\(3]          # coarse pattern for basic text emojis
                | [\U0001F100-\U0001FFFF]  # coarse code range for unicode emojis
```

```
                )
                """, re.VERBOSE)

def clean_tokenize(text):
    # remove punctuation characters
    text = ''.join([char for char in text if char not in punctuation])
    # fold to lowercase
    text = text.lower()
    # remove stopwords
    text = ' '.join([word for word in text.split() if word.lower() not in sw])
    # remove extra white spaces
    text = re.sub(r'\s+', ' ', text)
    # strip the text
    text = text.strip()
    return(text)
    # could use this if we wanted to tokenize now
    # return RE_TOKEN.findall(text)
```

[ ]:
```
convention_db = sqlite3.connect("2020_Conventions.db")
convention_cur = convention_db.cursor()
```

0.1.1 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.

[ ]:
```
convention_data = []

# fill this list up with items that are themselves lists. The
# first element in the sublist should be the cleaned and tokenized
# text in a single string. The second element should be the party.

query_results = convention_cur.execute("SELECT * FROM conventions")


for row in query_results :
    text = row[5]  # convention text is in the 6th column
    party = row[0]  # political party is in the 1st column

    # Preprocess the text using the clean_tokenize function
    cleaned_text = clean_tokenize(text)

    # Append tokenized_text and party as a sublist to convention_data
    convention_data.append([cleaned_text, party])
```

Let's look at some random entries and see if they look right.

```
[ ]: random.choices(convention_data,k=10)
```

```
[ ]: [['thanks bernie want thank joining us segment mean sincerely honor run there's
     even greater honor stand support joe biden kamala harris',
       'Democratic'],
      ['singing', 'Democratic'],
      ['thank mr president honor mine', 'Republican'],
      ['daughter's murder media didn't seem interested facts found learned gun
     control laws didn't fail daughter people gunman threatened kill classmates
     threatened rape threatened shoot school every red flag could imagine school
     didn't miss red flags knowingly ignored far left democrats school district made
     shooting possible implemented something called restorative justice policy really
     blames teachers student's failures puts kids teachers risk make shootings likely
     built pioneering approach discipline safety fine old approach discipline safety
     called discipline safety obamabiden administration took parkland's bad policies
     forced schools across america',
       'Republican'],
      ['south bend feared best days behind us reimagined economy new jobs even new
     industries hoosier state ready lead america's recovery diverse communities
     talented workers best world agriculture joe biden's plan gives us blueprint
     revitalize industrial cities rural areas alike indiana casts 2 votes friend
     bernie sanders 86 votes next president joe biden',
       'Democratic'],
      ['visiting parents grandparents window nursing home worrying time they'll get
     sick',
       'Democratic'],
      ['acting secretary wolf present five candidates naturalization representing
     five countries',
       'Republican'],
      ['it's rare quality bring empathy skills process governing joe biden never
     forgets that's point moving wheels government',
       'Democratic'],
      ['never allow mob rule strongest possible terms republican party condemns
     rioting looting arson violence seen democratrun cities like kenosha minneapolis
     portland chicago new york many others democrat run violence danger streets many
     democratrun cities throughout america problem could easily fixed wanted call
     we're ready go we'll take care problem matter hours call wait call it's bad wait
     call must always law order federal crimes investigated prosecuted punished
     fullest extent law',
       'Republican'],
      ['led joe biden it's aimed ending disease killed vice president's son 600000
     americans every year',
       '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.

[ ]:
```
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 2391 as features in the model.
```

[ ]:
```
def conv_features(text,fw) :
    """Given some text, this returns a dictionary holding the
       feature words.

       Args:
            * 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}

    """

    # Split the text into individual words
    words = text.split()

    # Create a dictionary to store feature words found in the text
    ret_dict = {word: True for word in words if word in fw}

    return (ret_dict)
```

[ ]:
```
# use assertions to test that the code is working as intended
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.

[ ]:
```
featuresets = [(conv_features(text,feature_words), party) for (text, party) in␣
 ↪convention_data]
```

[ ]:
```
random.seed(20220507)
random.shuffle(featuresets)

test_size = 500
```

[ ]:
```
test_set, train_set = featuresets[:test_size], featuresets[test_size:]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))
```

```
0.5
```

[ ]:
```
test_party_counts = Counter(party for _, party in test_set)

# Print the counts
for party, count in test_party_counts.items():
    print(f"Party: {party}, Count: {count}")
```

```
Party: Democratic, Count: 308
Party: Republican, Count: 192
```

[ ]:
```
classifier.show_most_informative_features(25)
```

```
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
                 freedoms = True           Republ : Democr =     18.2 : 1.0
                  climate = True           Democr : Republ =     17.8 : 1.0
                 supports = True           Republ : Democr =     17.1 : 1.0
                    crime = True           Republ : Democr =     16.1 : 1.0
                    media = True           Republ : Democr =     14.9 : 1.0
                  beliefs = True           Republ : Democr =     13.0 : 1.0
                countries = True           Republ : Democr =     13.0 : 1.0
                  defense = True           Republ : Democr =     13.0 : 1.0
```

```
          isis = True           Republ : Democr =     13.0 : 1.0
       liberal = True           Republ : Democr =     13.0 : 1.0
      religion = True           Republ : Democr =     13.0 : 1.0
         trade = True           Republ : Democr =     12.7 : 1.0
          flag = True           Republ : Democr =     12.1 : 1.0
     greatness = True           Republ : Democr =     12.1 : 1.0
       abraham = True           Republ : Democr =     11.9 : 1.0
        defund = True           Republ : Democr =     11.9 : 1.0
          drug = True           Republ : Democr =     10.9 : 1.0
    department = True           Republ : Democr =     10.9 : 1.0
     destroyed = True           Republ : Democr =     10.9 : 1.0
         enemy = True           Republ : Democr =     10.9 : 1.0
     amendment = True           Republ : Democr =     10.3 : 1.0
```

0.1.2 My Observations

Of all the top 25 most informative features in the classifier, 23 of them strongly favor words appearing in a Republican convention. The only featured words with a high enough ratio in the other direction are "votes" and "climate". This could be a cause of the Republican party being more narrowly focused on what it deems as issues while the Democratic party is more broad in their approach. I believe this is likely to sway many of the predictions towards the Republican party when they should have instead been Democratic. This idea may be backed by the fact that the classifier only had an accuracy of 50% on the test despite the party breakdown being 60-40. Thus, a simple baseline model that predicts only Democratic would have even performed better.

0.2 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.

[ ]:
```
cong_db = sqlite3.connect("congressional_data.db")
cong_cur = cong_db.cursor()
```

[ ]:
```
results = cong_cur.execute("""
            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 = list(results) # Just to store it, since the query is time consuming
```

[ ]:
```
tweet_data = []

# Now fill up tweet_data with sublists like we did on the convention speeches.
# Note that this may take a bit of time, since we have a lot of tweets.

for row in results:
    tweet_text = row[2]  # tweet text is in the 3rd column
    party = row[1]  # political party is in the 2nd column

    cleaned_tweet = clean_tokenize(tweet_text.decode('utf-8'))  # Decode the␣
 ↪tweet text first, then tokenize

    tweet_data.append([cleaned_tweet, party])
```

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…

[ ]:
```
random.seed(20201015)

tweet_data_sample = random.choices(tweet_data,k=10)
```

[ ]:
```
for tweet, party in tweet_data_sample :
    estimated_party = classifier.classify(conv_features(tweet, feature_words))
    # Fill in the right-hand side above with code that estimates the actual␣
 ↪party

    print(f"Here's our (cleaned) tweet: {tweet}")
    print(f"Actual party is {party} and our classifer says {estimated_party}.")
    print("")
```

```
Here's our (cleaned) tweet: icymi saturday joined recent graduates us merchant
marine academy helped welcome aboard new group plebes grateful young men women
answered call serve httpstcoevee3hhdfw
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: istand men women law enforcement sheriffclarke
sheriffclarkchallenge 1000 ppl ×25maga httpstconluvnsv9km httpstcoc6vmgb1hvn
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: httptcoclphxnpdjx
Actual party is Republican and our classifer says Democratic.

Here's our (cleaned) tweet: sfljetsfanclub jetsfanclub see new fp mag piece
nyjets player speaks extreme anti israel conf httptco0wesp7fwlx
```

```
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: simply put trump family separation policychild abuse
long term detention military facilities new trump executive orderchild abuse
Actual party is Democratic and our classifer says Republican.

Here's our (cleaned) tweet: everyday policy personnel wh shows us exactly values
priorities -this november let’s show yearofthewoman bluewave httpstcobmctaqmoql
Actual party is Democratic and our classifer says Republican.

Here's our (cleaned) tweet: clermont senior club hosted listening session
community members could share thoughts represent httptcotfrgfrpwdt
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: looking fun safe halloween helpful safety tips txdps
httpstcoafrr75zjcp
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: stand 2nd amendment rights sign petition today
httpstcobotu9nqkyt httpstcovo6jfzwixu
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: proud endorsed aftkc represents educators across
kansas city metro area together show repsamgraves washington hard working people
like matter missourimatters httpstcos1l2igcoj6
Actual party is Democratic and our classifer says Republican.
```

Now that we’ve looked at it some, let’s score a bunch and see how we’re doing.

[ ]:
```python
parties = ['Republican','Democratic']
results = defaultdict(lambda: defaultdict(int))

for p in parties :
    for p1 in parties :
        results[p][p1] = 0


num_to_score = 10000
random.shuffle(tweet_data)

for idx, tp in enumerate(tweet_data) :
    tweet, party = tp

    # get the estimated party
    estimated_party = classifier.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': 3801, 'Democratic': 613}),
             'Democratic': defaultdict(int,
                        {'Republican': 4723, 'Democratic': 865})})
```

[ ]:
```
print("Total tweets by political party:")
tweet_party_counts = Counter(sublist[1] for sublist in tweet_data)
total_instances = sum(tweet_party_counts.values())

# Print the counts and percentage breakdown
for party, count in tweet_party_counts.items():
    percentage = (count / total_instances) * 100
    print(f"Party: {party}, Count: {count}, Percentage: {percentage:.2f}%")
```

```
Total tweets by political party:
Party: Republican, Count: 288531, Percentage: 43.41%
Party: Democratic, Count: 376125, Percentage: 56.59%
```

[ ]:
```
word_counts = defaultdict(list)

# Calculate word counts for each text and store them by party
for text, party in convention_data:
    words = clean_tokenize(text)
    word_counts[party].append(len(words))

average_word_counts = {party: sum(counts) / len(counts) for party, counts in␣
 ↪word_counts.items()}

for party, average_count in average_word_counts.items():
    print(f"Party: {party}, Average Word Count: {average_count:.2f}")
```

```
Party: Democratic, Average Word Count: 178.53
Party: Republican, Average Word Count: 328.16
```

0.2.1 Reflections

As predicted in the previous section, our Naive Bayes classifier overpredicts the Republican class and underpredicts the Democratic one. Of the true Republican instances, 86.1% of the estimated classes were correct. However, the model also predicted 84.5% of the true Democratic tweets as being Republican. Overall, this is an accuracy score of only 46.65% which means that it does not perform well against an all-democratic baseline and even would not outperfrom a model that

predicted the exact opposite results. Because of this imbalance, I also wanted to see how many words each party's convention text had to determine if this was part of the class imbalance prediction problem. As we can see, the average length of text for the Republican convention data was almost double that of the Democratic convention text which leads to larger featuring of words.