Political Naive Bayes

September 26, 2024

0.1 Ravita Kartawinata

Ravita's Git

0.2 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. You can download the required DB from the shared dropbox or from blackboard

[1]:
```python
import sqlite3
import nltk
import random
import numpy as np
from collections import Counter, defaultdict

# Feel free to include your text patterns functions
#from text_functions_solutions import clean_tokenize, get_patterns
```

[2]:
```python
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import make_pipeline
import warnings
warnings.filterwarnings("ignore")
from sklearn.metrics import accuracy_score, classification_report
```

[3]:
```python
convention_db = sqlite3.connect("/users/rkartawi/Desktop/Ravita/MSADS/509/Mod4/
  ↪2020_Conventions.db")
convention_cur = convention_db.cursor()
```

[4]:
```python
# # query to get the tables and colums name for next query
# # Chatgpt
# convention_cur.execute("SELECT name FROM sqlite_master WHERE type='table';")
# tables = convention_cur.fetchall()
# # Get table names and their columns
# for table_name in tables:
#     table_name = table_name[0]
#     print(f"\nTable: {table_name}")

#     convention_cur.execute(f"PRAGMA table_info({table_name});")
#     schema = convention_cur.fetchall()

#     for column in schema:
#         print(column)
```

0.2.1 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.

[5]:
```
stop_words = set(stopwords.words('english'))

convention_data = []

query_results = convention_cur.execute("SELECT text, party FROM conventions")

for row in query_results:
    # store the results in convention_data
    text = row[0]
    party = row[1]

    # Cleaning processing
    tokens = word_tokenize(text.lower())
    cleaned_text = ' '.join([word for word in tokens if word.isalpha() and word␣
 ↪not in stop_words])

    convention_data.append([cleaned_text, party])

# for item in convention_data:
#     print("Cleaned Text:", item[0])
#     print("Cleaned Party:", item[1])
#     print("-" * 50)
```

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

[6]:
```
random.choices(convention_data,k=5)
```

[6]:
```
[['elect taxes going raised cut', 'Republican'],
 ['state vermont strongly believing economic justice social justice racial
justice environmental justice proudly supporting democracy constitution united
states vehemently opposed authoritarianism racism trump administration proud
cast votes vermont senator bernie sanders votes next president united states joe
biden',
  'Democratic'],
 ['singing', 'Democratic'],
 ['paying every worker fair wage', 'Democratic'],
 ['simply endure recession struggle survive working men women america get
crushed yet time hand government washed career politician nothing puppet radical
left democrats lifelong resident wisconsin fan badger football team many may
realize wisconsin badgers president share three common qualities smart tough
dependable businessman tell qualities need country leader need reelect donald
trump thank god bless america',
  'Republican']]
```

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.

[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.")
# print("Feature words:", feature_words)
```

```
With a word cutoff of 5, we have 2236 as features in the model.
```

[8]:
```
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}

    """
    ret_dict = dict()
    words = text.split()

    # if in feature words (fw), add it to the dictionary
    for word in words:
        if word in fw:
            ret_dict[word] = True
    return(ret_dict)
```

[9]:
```
assert(len(feature_words)>0)
assert(conv_features("biden is the president",feature_words)==
       {'biden':True,'president':True})
assert(conv_features("immigrant in america should be citizens",feature_words)==
                     {'immigrant':True,'america':True,"citizens":True})
```

[10]:
```
assert(len(feature_words)>0)
assert(conv_features("donald is the president",feature_words)==
       {'donald':True,'president':True})
assert(conv_features("some people in america are citizens",feature_words)==
                     {'people':True,'america':True,"citizens":True})
```

The assertions are successful for both parties. It indicates that the conv_features function is correctly identifying and returning the relevant feature words from conventions/text

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.

[11]:
```
featuresets = [(conv_features(text,feature_words), party) for (text, party) in␣
↪convention_data]
```

[12]:
```
random.seed(20220507)
random.shuffle(featuresets)

test_size = 500
```

[13]:
```
test_set, train_set = featuresets[:test_size], featuresets[test_size:]
classifier = nltk.NaiveBayesClassifier.train(train_set)
print(nltk.classify.accuracy(classifier, test_set))
```

```
0.494
```

[14]:
```
classifier.show_most_informative_features(25)
```

```
Most Informative Features
                   china = True           Republ : Democr =     27.1 : 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 =     15.8 : 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
                  defund = 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
                    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
```

[15]:
```
# calculate random guessing with most common party and count
party_counts = Counter(party for text, party in convention_data)
most_common_party, most_common_count = party_counts.most_common(1)[0]
total_count = len(convention_data)

# accuracy of random guessing
random_guess_accuracy = most_common_count / total_count
print(f"Accuracy of random guessing: {random_guess_accuracy:.2f}")
```

```
Accuracy of random guessing: 0.61
```

Write a little prose here about what you see in the classifier. Anything odd or interesting?

0.2.2 My Observations

The Naive Bayes classifier's accuracy of 49.4% is notably lower than the random guessing accuracy of 61%, this indicates that the model is struggling to identify meaningful patterns, which is surprising given that Naive Bayes is typically a robust choice for text classification tasks. The disparity suggests that the chosen features may not adequately capture the complexities of the political speeches in the dataset.

Although, the analysis of informative features shows that certain words strongly correlate with specific parties, yet the model still fails to outperform random guessing. This discrepancy might point to issues in the data preprocessing stage, such as inadequate handling of stopwords or overly restrictive feature selection criteria. Some topics, such as "climate" and "china," shows potential richness in the data that is not being utilized. It may also be beneficial to experiment with different classification algorithms, as alternatives like Support Vector Machines or Random Forests might yield better results. In summary, these findings highlight a critical need for further exploration of both feature engineering and model selection to enhance classification accuracy in political discourse analysis.

0.3 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.

[16]:
```
cong_db = sqlite3.connect("/users/rkartawi/Desktop/Ravita/MSADS/509/Mod4/
 ↪congressional_data.db")
cong_cur = cong_db.cursor()
```

[17]:
```
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
```

[18]:
```
tweet_data = []

for row in results:
    party = row[1]
    tweet_text = row[2]

    if isinstance(tweet_text, bytes):
        tweet_text = tweet_text.decode('utf-8')

    # Cleaning process
    tweet_text = re.sub(r'http\S+|www\S+|https\S+', '', tweet_text, flags=re.
 ↪MULTILINE)  # Remove URLs Regex
    tokens = word_tokenize(tweet_text.lower())
    cleaned_text = ' '.join([word for word in tokens if word.isalpha() and word␣
 ↪not in stop_words])  # Remove non-alphabetic and stop words

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

[19]:
```
tweet_df = pd.DataFrame(tweet_data, columns=['cleaned_text', 'party'])

X = tweet_df['cleaned_text']
y = tweet_df['party']

vectorizer = TfidfVectorizer()
X_tfidf = vectorizer.fit_transform(X)
```

[20]:
```
X_train, X_test, y_train, y_test = train_test_split(X_tfidf, y, test_size=0.2,␣
 ↪random_state=42)

# classifier = LogisticRegression()
# classifier.fit(X_train, y_train)

classifier = MultinomialNB()
classifier.fit(X_train, y_train)
```

[20]:
```
MultinomialNB()
```

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…

[30]:
```
random.seed(20201014)
random.shuffle(tweet_data)

tweet_data_sample = random.choices(tweet_data,k=10)
for tweet, party in tweet_data_sample:
    # Fill in the right-hand side above with code that estimates the actual␣
 ↪party
    X_sample = vectorizer.transform([tweet])
    estimated_party = classifier.predict(X_sample)[0]

    print(f"Here's our (cleaned) tweet: {tweet}")
    match = ' ' if party == estimated_party else ' '
    print(f"Actual party is {party} and our classifier says {estimated_party}.␣
 ↪{match}")
    print("")
```

```
Here's our (cleaned) tweet: met w angie settle wvhealthright charleston make
healthcare accessible thousands west virginians every year
```

```
Actual party is Republican and our classifier says Democratic.

Here's our (cleaned) tweet: absolute privilege recognize cpl rosser service
country medal honor recipient
Actual party is Republican and our classifier says Democratic.

Here's our (cleaned) tweet: time put america first deal iran put america harm
way postnewstxcity
Actual party is Republican and our classifier says Republican.

Here's our (cleaned) tweet: fernando exactly kind young person america embracing
alone across nation dreamers contributing amp strengthening communities america
afford republicans continue push away
Actual party is Democratic and our classifier says Democratic.

Here's our (cleaned) tweet: vp middle class joe biden honor got ta love joe
Actual party is Democratic and our classifier says Democratic.

Here's our (cleaned) tweet: washingtonpost editorial reopen white house doors
tourists
Actual party is Republican and our classifier says Democratic.

Here's our (cleaned) tweet: americans stand solidarity afghan people following
heinous attack carried ramadan
Actual party is Democratic and our classifier says Democratic.

Here's our (cleaned) tweet: unhinged overly emotional shrill amp needs smile
believe kavanaughhearings
Actual party is Democratic and our classifier says Democratic.

Here's our (cleaned) tweet: capitalism trial week president obama went whirlwind
tour wall street
Actual party is Republican and our classifier says Republican.

Here's our (cleaned) tweet: keep pressure potus executive order must call muslim
ban
Actual party is Democratic and our classifier says Democratic.
```

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

[24]:
```python
y_pred = classifier.predict(X_test)

# performance
accuracy = accuracy_score(y_test, y_pred)
print(f"Naive Bayes accuracy: {accuracy:.2f}")
```

```
report = classification_report(y_test, y_pred, target_names=['Democratic',␣
 ↪'Republican'])
print("Classification Rpt:")
print(report)
```

```
Naive Bayes accuracy: 0.79
Classification Rpt:
              precision    recall  f1-score   support

  Democratic       0.77      0.89      0.82     74865
  Republican       0.82      0.66      0.73     58067

    accuracy                           0.79    132932
   macro avg       0.79      0.77      0.78    132932
weighted avg       0.79      0.79      0.78    132932
```

[25]:
```
# 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] = 0

num_to_score = 10000
random.shuffle(tweet_data)

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

    # Transform the cleaned tweet to the vector
    tweet_tfidf = vectorizer.transform([tweet])

    # Do the same thing as above and Store
    estimated_party = classifier.predict(tweet_tfidf)[0]
    results[party][estimated_party] += 1

    if idx >= num_to_score:
        break
```

[26]:
```
print("\nCount of Actual vs Estimated Party for 10000 samples:")
for actual_party in parties:
    print(f"\nActual Party: {actual_party}")
    for estimated_party in parties:
        print(f"  Estimated Party: {estimated_party} - Count:␣
↪{results[actual_party][estimated_party]}")
```

```
Count of Actual vs Estimated Party for 10000 samples:

Actual Party: Republican
  Estimated Party: Republican - Count: 3013
  Estimated Party: Democratic - Count: 1335

Actual Party: Democratic
  Estimated Party: Republican - Count: 536
  Estimated Party: Democratic - Count: 5117
```

0.3.1 Reflections

The Naive Bayes classifier achieved an overall accuracy of 79%, it means it correctly classified approximately four out of five tweets. The classification report highlights that the precision for predicting Democratic tweets is 77%, with a recall of 89%, suggesting the model is effective at identifying Democratic tweets but may occasionally misclassify Republican tweets as Democratic. In contrast, the precision for Republican tweets is higher at 82%, but the recall is lower at 66%, indicating that while the model is generally good at identifying Republican tweets, it misses a significant portion of them.

The confusion matrix only reveals 10,000 tweets, the model classification report has different result due to the different dataset. In this sample, 30.53% of Republican tweets correctly while misclassifying 13.68% as Democratic, and correctly identified 50.60% of Democratic tweets while misclassifying 5.20% as Republican. One improvements that can be made is particularly in increasing the recall for Republican tweets to reduce misclassification.

In conclusion, although the speech data might not be sufficient in predicting the party, however having additional dataset (i.e: tweet), different feature engineering and algorithm, it could change the overall performance accuracy.

0.3.2 References:

Dib, F. (n.d.). Build, test, and debug regex. regex101. https://regex101.com/

OpenAI. (2024). ChatGPT (September 24 version) [Large language model]. https://chat.openai.com/

Chandler, J. (n.d.). 37chandler/TM-NB-Conventions. GitHub. https://github.com/37chandler/tm-nb-