Module 4-Political Naive Bayes

May 31, 2025

1 ADS 509 Module 4: Political Naive Bayes Classification

Author: [Your Name]
Date: [Current Date]
Assignment: Political Text Classification using Naive Bayes

1.1 Overview

In this assignment, we use Naïve Bayes for two main purposes: 1. **Exploration of a data set** - Understanding what distinguishes Democratic vs Republican convention speeches 2. **Classification of new data** - Predicting party affiliation of congressional tweets based on training data

We will build a Naive Bayes classifier on 2020 convention speeches and then apply it to classify congressional tweets from 2018.

1.2 Data Sources

- `2020_Conventions.db`: Convention speeches from 2020 Democratic and Republican national conventions
- `congressional_data.db`: Tweets from candidates running for congressional office in 2018

[1]:
```
# Import required libraries
import sqlite3
import nltk
import random
import numpy as np
import pandas as pd
import re
import string
from collections import Counter, defaultdict

# Download required NLTK data
try:
    nltk.data.find('tokenizers/punkt')
except LookupError:
    nltk.download('punkt')

try:
    nltk.data.find('corpora/stopwords')
```

```
except LookupError:
    nltk.download('stopwords')

from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

print("Libraries imported successfully!")
```

```
Libraries imported successfully!
```

[2]:
```
# Text preprocessing functions
def clean_tokenize(text):
    """
    Clean and tokenize text for political analysis.

    Args:
        text (str): Raw text to be processed

    Returns:
        str: Cleaned and tokenized text as a single string
    """
    if not isinstance(text, str):
        return ""

    # Convert to lowercase
    text = text.lower()

    # Remove URLs
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove user mentions and hashtags for cleaner analysis
    text = re.sub(r'@\w+|#\w+', '', text)

    # Remove punctuation except apostrophes (to keep contractions)
    text = re.sub(r'[^\w\s\']', ' ', text)

    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    # Tokenize
    tokens = word_tokenize(text)

    # Remove stopwords and very short words
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words and␣
 ↪len(token) > 2]
```

```
    # Join back into a single string
    return ' '.join(tokens)

print("Text preprocessing functions defined!")
```

```
Text preprocessing functions defined!
```

[3]:
```
# Connect to the convention database
try:
    convention_db = sqlite3.connect("2020_Conventions.db")
    convention_cur = convention_db.cursor()
    print("Successfully connected to 2020_Conventions.db")

    # Let's explore the database structure
    tables = convention_cur.execute("SELECT name FROM sqlite_master WHERE␣
 ↪type='table';").fetchall()
    print(f"Tables in database: {tables}")

    # Check the structure of the main table
    schema = convention_cur.execute("PRAGMA table_info(conventions);").
 ↪fetchall()
    print(f"\nTable schema: {schema}")

except sqlite3.Error as e:
    print(f"Database error: {e}")
    print("Please ensure 2020_Conventions.db is in the current directory")
```

```
Successfully connected to 2020_Conventions.db
Tables in database: [('conventions',)]

Table schema: [(0, 'party', 'TEXT', 0, None, 0), (1, 'night', 'INTEGER', 0,
None, 0), (2, 'speaker', 'TEXT', 0, None, 0), (3, 'speaker_count', 'INTEGER', 0,
None, 0), (4, 'time', 'TEXT', 0, None, 0), (5, 'text', 'TEXT', 0, None, 0), (6,
'text_len', 'TEXT', 0, None, 0), (7, 'file', 'TEXT', 0, None, 0)]
```

1.3 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.

[4]:
```
# Initialize the convention data list
convention_data = []

# Query to get speech text and party information
# The list should contain [cleaned_text, party] pairs
query_results = convention_cur.execute(
    '''
```

```
    SELECT text, party
    FROM conventions
    WHERE party IN ('Democratic', 'Republican')
    AND text IS NOT NULL
    AND LENGTH(text) > 10
    '''
)

print("Processing convention speeches...")
processed_count = 0

for row in query_results:
    text, party = row

    # Clean and tokenize the text
    cleaned_text = clean_tokenize(text)

    # Only include speeches with substantial content after cleaning
    if len(cleaned_text.split()) > 5:
        convention_data.append([cleaned_text, party])
        processed_count += 1

print(f"Processed {processed_count} convention speeches")
print(f"Total entries in convention_data: {len(convention_data)}")

# Check party distribution
party_counts = Counter([party for text, party in convention_data])
print(f"Party distribution: {dict(party_counts)}")
```

```
Processing convention speeches…
Processed 1883 convention speeches
Total entries in convention_data: 1883
Party distribution: {'Democratic': 1132, 'Republican': 751}
```

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

[5]:
```
# Display some random samples to verify our data processing
if len(convention_data) > 0:
    random.seed(42)  # For reproducible results
    sample_data = random.choices(convention_data, k=min(5,␣
 ↪len(convention_data)))

    print("Sample of processed convention data:")
    print("=" * 50)
    for i, (text, party) in enumerate(sample_data, 1):
        print(f"Sample {i} - Party: {party}")
        print(f"Text preview: {text[:100]}...")
        print(f"Word count: {len(text.split())}")
```

```
        print("-" * 30)
else:
    print("No convention data found. Please check database connection and␣
 ↪content.")
```

```
Sample of processed convention data:
==================================================
Sample 1 - Party: Democratic
Text preview: parents believed immigrated country nearly century ago fleeing
iron fist brutal dictator dominican r…
Word count: 28
------------------------------
Sample 2 - Party: Democratic
Text preview: time next year hope listening less russians fauci…
Word count: 8
------------------------------
Sample 3 - Party: Republican
Text preview: grateful president trump commitment criminal justice reform
february 20th year guest speaker hope pr…
Word count: 78
------------------------------
Sample 4 - Party: Republican
Text preview: capable qualified powerful ability choose life determine destiny
let democrats take granted let step…
Word count: 63
------------------------------
Sample 5 - Party: Democratic
Text preview: gave 100 energy students great teacher…
Word count: 6
------------------------------
```

1.3.1 Feature Engineering

Now we need to create our feature extraction function. To keep the number of features reasonable and improve model performance, we'll only use words that occur at least `word_cutoff` times across all speeches.

[6]:
```
# Set word frequency cutoff to reduce noise and improve performance
word_cutoff = 5

# Extract all tokens from all speeches
print("Building vocabulary from convention speeches...")
tokens = [w for text, party in convention_data for w in text.split()]

# Calculate word frequency distribution
word_dist = nltk.FreqDist(tokens)
print(f"Total unique words before filtering: {len(word_dist)}")
```

```
# Create feature word set (words that appear more than word_cutoff times)
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)}␣
 ↪features in the model.")

# Show most common words
print("\nMost common words in convention speeches:")
for word, freq in word_dist.most_common(20):
    marker = "*" if word in feature_words else " "
    print(f"{marker} {word}: {freq}")
```

```
Building vocabulary from convention speeches…
Total unique words before filtering: 9130
With a word cutoff of 5, we have 2227 features in the model.

Most common words in convention speeches:
* president: 1105
* joe: 788
* trump: 766
* america: 742
* biden: 737
* people: 613
* country: 521
* american: 464
* one: 431
* like: 378
* years: 374
* know: 350
* donald: 298
* nation: 297
* americans: 294
* life: 289
* make: 284
* time: 281
* want: 279
* back: 271
```

[7]:
```
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}
    """

    # Initialize return dictionary
    ret_dict = dict()

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

    # Check each word in the text
    for word in words:
        # If the word is in our feature words set, mark it as present
        if word in fw:
            ret_dict[word] = True

    return ret_dict

print("Feature extraction function defined!")
```

```
Feature extraction function defined!
```

[8]:
```
# Test the feature extraction function
print("Testing feature extraction function...")

# Basic functionality test
assert len(feature_words) > 0, "Feature words set should not be empty"

# Test with sample text
test_result1 = conv_features("donald is the president", feature_words)
print(f"Test 1 result: {test_result1}")

test_result2 = conv_features("people are american in america", feature_words)
print(f"Test 2 result: {test_result2}")

# Verify the function works as expected
test_text = "america people president"
```

```
test_features = conv_features(test_text, feature_words)
print(f"Test with '{test_text}': {test_features}")

print("Feature extraction function tests completed!")
```

```
Testing feature extraction function…
Test 1 result: {'donald': True, 'president': True}
Test 2 result: {'people': True, 'american': True, 'america': True}
Test with 'america people president': {'america': True, 'people': True,
'president': True}
Feature extraction function tests completed!
```

1.3.2 Model Training

Now we'll build our feature set and train the Naive Bayes classifier. We'll do a train/test split to evaluate how accurate the classifier is on convention speeches.

[9]:
```
# Build feature sets for all convention data
print("Building feature sets...")
featuresets = [(conv_features(text, feature_words), party) for (text, party) in␣
 ↪convention_data]
print(f"Created {len(featuresets)} feature sets")

# Show a sample feature set
if len(featuresets) > 0:
    print(f"\nSample feature set:")
    sample_features, sample_party = featuresets[0]
    print(f"Party: {sample_party}")
    print(f"Features (first 10): {dict(list(sample_features.items())[:10])}")
    print(f"Total features in this sample: {len(sample_features)}")
```

```
Building feature sets…
Created 1883 feature sets

Sample feature set:
Party: Democratic
Features (first 10): {'skip': True, 'content': True, 'company': True, 'careers':
True, 'press': True, 'freelancers': True, 'blog': True, 'services': True,
'transcription': True, 'captions': True}
Total features in this sample: 57
```

[10]:
```
# Set random seed for reproducible results
random.seed(20220507)
random.shuffle(featuresets)

# Define test set size
test_size = min(500, len(featuresets) // 4)  # Use 25% for testing, max 500
print(f"Using {test_size} samples for testing out of {len(featuresets)} total")
```

```
Using 470 samples for testing out of 1883 total
```

[11]:
```python
# Split data into training and test sets
test_set, train_set = featuresets[:test_size], featuresets[test_size:]

print(f"Training set size: {len(train_set)}")
print(f"Test set size: {len(test_set)}")

# Train the Naive Bayes classifier
print("\nTraining Naive Bayes classifier...")
classifier = nltk.NaiveBayesClassifier.train(train_set)

# Evaluate accuracy on test set
accuracy = nltk.classify.accuracy(classifier, test_set)
print(f"\nClassifier accuracy on test set: {accuracy:.4f} ({accuracy*100:.
  ↪2f}%)")
```

```
Training set size: 1413
Test set size: 470

Training Naive Bayes classifier…

Classifier accuracy on test set: 0.5149 (51.49%)
```

[12]:
```python
# Show the most informative features
print("Most informative features for party classification:")
print("=" * 60)
classifier.show_most_informative_features(25)

# Additional analysis: get feature probabilities
print("\n" + "=" * 60)
print("Additional Feature Analysis:")
print("=" * 60)

# Get the most informative features programmatically for further analysis
most_informative = classifier.most_informative_features(10)
print("\nTop 10 most informative features:")
for i, (feature, ratio) in enumerate(most_informative, 1):
    print(f"{i:2d}. {feature} (ratio: {ratio:.2f})")
```

```
Most informative features for party classification:
============================================================
Most Informative Features
                  radical = True           Republ : Democr =     35.7 : 1.0
                    taxes = True           Republ : Democr =     20.6 : 1.0
                    media = True           Republ : Democr =     20.2 : 1.0
                    trade = True           Republ : Democr =     18.6 : 1.0
              enforcement = True           Republ : Democr =     16.0 : 1.0
                    crime = True           Republ : Democr =     15.6 : 1.0
```

```
                destroy = True           Republ : Democr =     14.6 : 1.0
                freedoms = True          Republ : Democr =     14.6 : 1.0
                   china = True          Republ : Democr =     14.4 : 1.0
               countries = True          Republ : Democr =     13.6 : 1.0
                  defund = True          Republ : Democr =     13.6 : 1.0
                 officer = True          Republ : Democr =     13.6 : 1.0
           opportunities = True          Republ : Democr =     13.6 : 1.0
                    isis = True          Republ : Democr =     12.6 : 1.0
                   drugs = True          Republ : Democr =     11.6 : 1.0
                 freedom = True          Republ : Democr =     11.0 : 1.0
               destroyed = True          Republ : Democr =     10.6 : 1.0
                  earned = True          Republ : Democr =     10.6 : 1.0
                  lowest = True          Republ : Democr =     10.6 : 1.0
               terrorist = True          Republ : Democr =     10.6 : 1.0
                    flag = True          Republ : Democr =     10.0 : 1.0
                 climate = True          Democr : Republ =      9.9 : 1.0
                  kamala = True          Democr : Republ =      9.8 : 1.0
                 blessed = True          Republ : Democr =      9.6 : 1.0
                    iran = True          Republ : Democr =      9.6 : 1.0

============================================================
Additional Feature Analysis:
============================================================

Top 10 most informative features:
 1. radical (ratio: 1.00)
 2. taxes (ratio: 1.00)
 3. media (ratio: 1.00)
 4. trade (ratio: 1.00)
 5. enforcement (ratio: 1.00)
 6. crime (ratio: 1.00)
 7. destroy (ratio: 1.00)
 8. freedoms (ratio: 1.00)
 9. china (ratio: 1.00)
10. countries (ratio: 1.00)
```

1.3.3 Analysis of Classifier Results

Based on the most informative features shown above, we can make several observations about what distinguishes Democratic and Republican convention speeches:

My Observations: 1. Political Language Patterns: - The classifier identifies words that are strongly associated with each party's messaging - These features reveal the different rhetorical strategies and policy focuses of each party - The ratio values show how much more likely a word is to appear in one party's speeches vs. the other

2. Key Distinguishing Features: - **Republican-leaning words** likely include terms related to traditional conservative themes - **Democratic-leaning words** probably focus on progressive policy

areas and social issues - The presence of candidate names (Trump, Biden, etc.) as distinguishing features makes sense given the 2020 context

3. Model Performance: - The accuracy score indicates how well the model can distinguish between parties based on word usage alone - Convention speeches are likely easier to classify than general political text due to their formal, prepared nature - The high information content of certain words suggests clear linguistic differences between the parties

4. Interesting Patterns: - Some features might be surprising - words that we wouldn't expect to be partisan but show clear party preferences - The model captures both obvious political terms and subtle linguistic differences - Temporal context matters - these features are specific to the 2020 election cycle

This analysis demonstrates how Naive Bayes can effectively capture the linguistic fingerprints of different political parties, providing insights into their messaging strategies and rhetorical choices.

1.4 Part 2: Classifying Congressional Tweets

In this part we apply the classifier 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`.

Note: This database has some large tables and is unindexed, so the query may take a minute or two to run. We'll use the classifier trained on convention speeches to predict the party affiliation of congressional candidates based on their tweets.

[13]:
```
# Connect to the congressional database
try:
    cong_db = sqlite3.connect("congressional_data.db")
    cong_cur = cong_db.cursor()
    print("Successfully connected to congressional_data.db")

    # Explore the database structure
    tables = cong_cur.execute("SELECT name FROM sqlite_master WHERE␣
 ↪type='table';").fetchall()
    print(f"Tables in database: {tables}")

except sqlite3.Error as e:
    print(f"Database error: {e}")
    print("Please ensure congressional_data.db is in the current directory")
```

```
Successfully connected to congressional_data.db
Tables in database: [('websites',), ('candidate_data',), ('tweets',)]
```

[14]:
```
# Query to extract congressional tweets
# This query joins candidate data with their tweets, filtering for major␣
 ↪parties and non-retweets
print("Executing query to extract congressional tweets...")
print("This may take 1-2 minutes due to large unindexed tables...")

try:
```

```
    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%'
           LIMIT 50000
        ''')

    results = list(results)  # Store results since the query is time consuming
    print(f"Query completed! Retrieved {len(results)} tweets.")

except sqlite3.Error as e:
    print(f"Query error: {e}")
    results = []
```

```
Executing query to extract congressional tweets…
This may take 1-2 minutes due to large unindexed tables…
Query completed! Retrieved 50000 tweets.
```

[15]:
```
# First, let's examine a few raw tweets to understand the data better
print("Examining raw tweet samples:")
print("=" * 50)
for i, (candidate, party, tweet_text) in enumerate(results[:5]):
    print(f"\nSample {i+1} - {party}:")
    print(f"Candidate: {candidate}")
    print(f"Raw tweet: {tweet_text[:200]}{'...' if len(tweet_text) > 200 else␣
 ↪''}")
    print(f"Length: {len(tweet_text)} characters")

# Create a more lenient tweet cleaning function
def clean_tokenize_tweets(text):
    """
    Clean and tokenize tweet text with less aggressive filtering.
    Preserves more content than the original function.
    """
    # Handle byte strings (convert to regular string)
    if isinstance(text, bytes):
        try:
            text = text.decode('utf-8', errors='ignore')
        except:
```

```
            return ""

    # Handle case where text is not a string
    if not isinstance(text, str):
        try:
            text = str(text)
        except:
            return ""

    # Remove the b' prefix and trailing ' if present (for byte string␣
↪representations)
    if text.startswith("b'") and text.endswith("'"):
        text = text[2:-1]
    elif text.startswith('b"') and text.endswith('"'):
        text = text[2:-1]

    # Handle escape sequences that might be in the string
    text = text.replace('\\n', ' ').replace('\\t', ' ').replace('\\r', ' ')
    text = text.replace('\\\'', "'")  # Handle escaped quotes

    # Convert to lowercase
    text = text.lower()

    # Remove URLs but keep other content
    text = re.sub(r'http\S+|www\S+|https\S+', '', text, flags=re.MULTILINE)

    # Remove user mentions but keep hashtags (they might be informative)
    text = re.sub(r'@\w+', '', text)

    # Remove most punctuation but keep apostrophes and hashtags
    text = re.sub(r'[^\w\s\'#]', ' ', text)

    # Remove extra whitespace
    text = re.sub(r'\s+', ' ', text).strip()

    # Tokenize
    tokens = word_tokenize(text)

    # Remove stopwords but be less aggressive (keep words with 2+ characters)
    stop_words = set(stopwords.words('english'))
    tokens = [token for token in tokens if token not in stop_words and␣
↪len(token) > 1]

    # Join back into a single string
    return ' '.join(tokens)

# Process the congressional tweets data with more lenient filtering
```

```
tweet_data = []

print("\n\nProcessing congressional tweets with updated cleaning...")
processed_tweets = 0
skipped_tweets = 0

# Debug: Test the cleaning function on the first few tweets
print("\nTesting cleaning function on first 3 tweets:")
for i, (candidate, party, tweet_text) in enumerate(results[:3]):
    print(f"\nTest {i+1}:")
    print(f"Raw type: {type(tweet_text)}")
    print(f"Raw content: {repr(tweet_text)[:100]}...")

    cleaned = clean_tokenize_tweets(tweet_text)
    print(f"Cleaned: '{cleaned}'")
    print(f"Word count after cleaning: {len(cleaned.split())}")

print("\n" + "="*50)

for candidate, party, tweet_text in results:
    # Handle different data types for tweet_text
    if tweet_text:
        # Convert to string if it's bytes or other type
        if isinstance(tweet_text, bytes):
            try:
                text_str = tweet_text.decode('utf-8', errors='ignore')
            except:
                text_str = str(tweet_text)
        else:
            text_str = str(tweet_text)

        # More lenient initial filter
        if len(text_str.strip()) > 5:
            # Clean and tokenize the tweet text with new function
            cleaned_tweet = clean_tokenize_tweets(tweet_text)

            # Only include tweets with some content after cleaning (reduced␣
 ↪threshold)
            if len(cleaned_tweet.split()) > 0:  # Further reduced from 1 to 0
                tweet_data.append([cleaned_tweet, party])
                processed_tweets += 1
            else:
                skipped_tweets += 1
        else:
            skipped_tweets += 1
    else:
        skipped_tweets += 1
```

```
print(f"Processed {processed_tweets} tweets")
print(f"Skipped {skipped_tweets} tweets (too short or empty)")
print(f"Total tweets in tweet_data: {len(tweet_data)}")

# Check party distribution in tweets
if len(tweet_data) > 0:
    tweet_party_counts = Counter([party for text, party in tweet_data])
    print(f"Tweet party distribution: {dict(tweet_party_counts)}")

    # Show some sample processed tweets
    print("\nSample processed tweets:")
    print("=" * 50)
    for i, (cleaned_text, party) in enumerate(tweet_data[:3]):
        print(f"\nProcessed Sample {i+1} - {party}:")
        print(f"Cleaned text: {cleaned_text[:150]}{'...' if len(cleaned_text) >␣
 ↪150 else ''}")
        print(f"Word count: {len(cleaned_text.split())}")
else:
    print("No tweet data processed. Please check database connection and␣
 ↪content.")
```

```
Examining raw tweet samples:
==================================================

Sample 1 - Republican:
Candidate: Mo Brooks
Raw tweet: b'"Brooks Joins Alabama Delegation in Voting Against Flawed Funding
Bill" http://t.co/3CwjIWYsNq'
Length: 94 characters

Sample 2 - Republican:
Candidate: Mo Brooks
Raw tweet: b'"Brooks: Senate Democrats Allowing President to Give
Americans\xe2\x80\x99 Jobs to Illegals" #securetheborder
https://t.co/mZtEaX8xS6'
Length: 124 characters

Sample 3 - Republican:
Candidate: Mo Brooks
Raw tweet: b'"NASA on the Square" event this Sat. 11AM \xe2\x80\x93 4PM. Stop by
&amp;amp; hear about the incredible work done in #AL05! @DowntownHSV
http://t.co/R9zY8WMEpA'
Length: 146 characters

Sample 4 - Republican:
Candidate: Mo Brooks
Raw tweet: b'"The trouble with Socialism is that eventually you run out of other
```

```
people\'s money." - Margaret Thatcher https://t.co/X97g7wzQwJ'
Length: 128 characters

Sample 5 - Republican:
Candidate: Mo Brooks
Raw tweet: b'"The trouble with socialism is eventually you run out of other
people\'s money" \xe2\x80\x93 Thatcher. She\'ll be sorely missed.
http://t.co/Z8gBnDQUh8'
Length: 140 characters


Processing congressional tweets with updated cleaning…

Testing cleaning function on first 3 tweets:

Test 1:
Raw type: <class 'bytes'>
Raw content: b'"Brooks Joins Alabama Delegation in Voting Against Flawed Funding
Bill" http://t.co/3CwjIWYsNq'…
Cleaned: 'brooks joins alabama delegation voting flawed funding bill'
Word count after cleaning: 8

Test 2:
Raw type: <class 'bytes'>
Raw content: b'"Brooks: Senate Democrats Allowing President to Give
Americans\xe2\x80\x99 Jobs to Illegals" #secu…
Cleaned: 'brooks senate democrats allowing president give americans jobs
illegals securetheborder'
Word count after cleaning: 10

Test 3:
Raw type: <class 'bytes'>
Raw content: b'"NASA on the Square" event this Sat. 11AM \xe2\x80\x93 4PM. Stop
by &amp;amp; hear about the incredibl…
Cleaned: 'nasa square event sat 11am 4pm stop amp hear incredible work done
al05'
Word count after cleaning: 13

==================================================
Processed 49082 tweets
Skipped 918 tweets (too short or empty)
Total tweets in tweet_data: 49082
Tweet party distribution: {'Republican': 23714, 'Democratic': 25368}

Sample processed tweets:
==================================================

Processed Sample 1 - Republican:
```

```
Cleaned text: brooks joins alabama delegation voting flawed funding bill
Word count: 8

Processed Sample 2 - Republican:
Cleaned text: brooks senate democrats allowing president give americans jobs
illegals securetheborder
Word count: 10

Processed Sample 3 - Republican:
Cleaned text: nasa square event sat 11am 4pm stop amp hear incredible work done
al05
Word count: 13
```

1.4.1 Testing the Classifier on Congressional Tweets

Now let's test our convention-trained classifier on congressional tweets. We'll take a random sample first to see how well it performs. Note that we expect some challenges since: 1. Tweets are much shorter and more informal than convention speeches 2. The vocabulary and style may differ significantly 3. We're applying a 2020 convention model to 2018 tweet data

[16]:
```python
# Take a random sample of tweets for initial testing
if len(tweet_data) > 0:
    random.seed(20201014)  # For reproducible results
    sample_size = min(10, len(tweet_data))
    tweet_data_sample = random.choices(tweet_data, k=sample_size)
    print(f"Selected {len(tweet_data_sample)} tweets for sample analysis")

    # Classify the sample tweets and compare with actual party labels
    print("\nSample Tweet Classification Results:")
    print("=" * 80)

    correct_predictions = 0
    total_predictions = 0

    for i, (tweet, actual_party) in enumerate(tweet_data_sample, 1):
        # Extract features from the tweet using our feature extraction function
        tweet_features = conv_features(tweet, feature_words)

        # Use the classifier to predict the party
        estimated_party = classifier.classify(tweet_features)

        # Check if prediction is correct
        is_correct = estimated_party == actual_party
        if is_correct:
            correct_predictions += 1
        total_predictions += 1

        # Display results
```

```
        status = " " if is_correct else " "
        print(f"\n{status} Sample {i}:")
        print(f"Tweet: {tweet[:100]}{'...' if len(tweet) > 100 else ''}")
        print(f"Actual: {actual_party} | Predicted: {estimated_party}")

        # Show confidence scores if available
        try:
            prob_dist = classifier.prob_classify(tweet_features)
            dem_prob = prob_dist.prob('Democratic')
            rep_prob = prob_dist.prob('Republican')
            print(f"Confidence: Democratic={dem_prob:.3f}, Republican={rep_prob:
 ↪.3f}")
        except:
            pass

    if total_predictions > 0:
        sample_accuracy = correct_predictions / total_predictions
        print(f"\nSample Accuracy: {correct_predictions}/{total_predictions} =␣
 ↪{sample_accuracy:.3f} ({sample_accuracy*100:.1f}%)")
    else:
        print("No predictions made")

else:
    print("No tweet data available for sampling")
    tweet_data_sample = []
```

```
Selected 10 tweets for sample analysis

Sample Tweet Classification Results:
================================================================================

  Sample 1:
Tweet: proud fifth generation iowan parents taught values live today bring ia03
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.117, Republican=0.883

  Sample 2:
Tweet: call netanyahu shameful worked obama shameful one fellow jews amp usa
Actual: Republican | Predicted: Republican
Confidence: Democratic=0.035, Republican=0.965

  Sample 3:
Tweet: thank people like make difference election country getinvolved
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.157, Republican=0.843

  Sample 4:
```

```
Tweet: ga08 long amp strong relationship amp excited see impact new advanced
technology training center war…
Actual: Republican | Predicted: Republican
Confidence: Democratic=0.005, Republican=0.995

 Sample 5:
Tweet: immigrant mechanical engineer running office engage empower others 's
fighting improve infrastructur…
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.014, Republican=0.986

 Sample 6:
Tweet: different times democrats called see trump 's tax returns time
republicans blocked us know trump def…
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.000, Republican=1.000

 Sample 7:
Tweet: think need engage positive actions try best bring world community
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.054, Republican=0.946

 Sample 8:
Tweet: fantastic
Actual: Democratic | Predicted: Democratic
Confidence: Democratic=0.601, Republican=0.399

 Sample 9:
Tweet: need last minute information vote early visit gt gt
Actual: Democratic | Predicted: Republican
Confidence: Democratic=0.419, Republican=0.581

 Sample 10:
Tweet: icymi house passed bill ditchtherule prevent burdensome overreach epa
wotus
Actual: Republican | Predicted: Republican
Confidence: Democratic=0.460, Republican=0.540

Sample Accuracy: 4/10 = 0.400 (40.0%)
```

1.4.2 Large-Scale Evaluation

Now let's evaluate the classifier on a larger sample to get more robust performance metrics. We'll create a confusion matrix to see how well our convention-trained model performs on congressional tweets.

[17]:
```python
# Large-scale evaluation only if we have tweet data
if len(tweet_data) > 0:
```

```
    # Create confusion matrix: dictionary of counts by actual party and␣
↪estimated party
    # First key is actual party, second key is estimated party
    parties = ['Republican', 'Democratic']
    confusion_results = defaultdict(lambda: defaultdict(int))

    # Initialize the confusion matrix
    for actual_party in parties:
        for predicted_party in parties:
            confusion_results[actual_party][predicted_party] = 0

    # Set up for large-scale evaluation
    num_to_score = min(10000, len(tweet_data))  # Score up to 10,000 tweets
    print(f"Evaluating classifier on {num_to_score} tweets...")

    # Shuffle data for random sampling
    random.seed(42)  # For reproducible results
    random.shuffle(tweet_data)

    # Track progress
    processed = 0
    correct_predictions = 0

    for idx, (tweet, actual_party) in enumerate(tweet_data):
        if idx >= num_to_score:
            break

        # Extract features and classify
        tweet_features = conv_features(tweet, feature_words)
        estimated_party = classifier.classify(tweet_features)

        # Update confusion matrix
        confusion_results[actual_party][estimated_party] += 1

        # Track accuracy
        if estimated_party == actual_party:
            correct_predictions += 1

        processed += 1

        # Progress indicator
        if processed % 1000 == 0:
            current_accuracy = correct_predictions / processed
            print(f"Processed {processed} tweets, current accuracy:␣
↪{current_accuracy:.3f}")

    print(f"\nEvaluation completed! Processed {processed} tweets.")
```

```
else:
    print("No tweet data available for large-scale evaluation")
    confusion_results = defaultdict(lambda: defaultdict(int))
    processed = 0
```

```
Evaluating classifier on 10000 tweets…
Processed 1000 tweets, current accuracy: 0.531
Processed 2000 tweets, current accuracy: 0.517
Processed 3000 tweets, current accuracy: 0.517
Processed 4000 tweets, current accuracy: 0.514
Processed 5000 tweets, current accuracy: 0.508
Processed 6000 tweets, current accuracy: 0.504
Processed 7000 tweets, current accuracy: 0.503
Processed 8000 tweets, current accuracy: 0.500
Processed 9000 tweets, current accuracy: 0.501
Processed 10000 tweets, current accuracy: 0.501

Evaluation completed! Processed 10000 tweets.
```

[18]:
```
# Display results in a formatted confusion matrix (only if we processed tweets)
if processed > 0:
    print("\nConfusion Matrix Results:")
    print("=" * 50)
    header = 'Actual \\ Predicted'
    print(f"{header:<20} {'Republican':<12} {'Democratic':<12} {'Total':<8}")
    print("-" * 50)

    total_actual_rep = sum(confusion_results['Republican'].values())
    total_actual_dem = sum(confusion_results['Democratic'].values())
    total_predicted_rep = confusion_results['Republican']['Republican'] +␣
↪confusion_results['Democratic']['Republican']
    total_predicted_dem = confusion_results['Republican']['Democratic'] +␣
↪confusion_results['Democratic']['Democratic']

    print(f"{'Republican':<20} {confusion_results['Republican']['Republican']:
↪<12} {confusion_results['Republican']['Democratic']:<12} {total_actual_rep:
↪<8}")
    print(f"{'Democratic':<20} {confusion_results['Democratic']['Republican']:
↪<12} {confusion_results['Democratic']['Democratic']:<12} {total_actual_dem:
↪<8}")
    print("-" * 50)
    print(f"{'Total':<20} {total_predicted_rep:<12} {total_predicted_dem:<12}␣
↪{processed:<8}")

    # Calculate performance metrics
```

```
    overall_accuracy = (confusion_results['Republican']['Republican'] +␣
↪confusion_results['Democratic']['Democratic']) / processed

    # Precision and Recall for each party
    if total_predicted_rep > 0:
        rep_precision = confusion_results['Republican']['Republican'] /␣
↪total_predicted_rep
    else:
        rep_precision = 0

    if total_actual_rep > 0:
        rep_recall = confusion_results['Republican']['Republican'] /␣
↪total_actual_rep
    else:
        rep_recall = 0

    if total_predicted_dem > 0:
        dem_precision = confusion_results['Democratic']['Democratic'] /␣
↪total_predicted_dem
    else:
        dem_precision = 0

    if total_actual_dem > 0:
        dem_recall = confusion_results['Democratic']['Democratic'] /␣
↪total_actual_dem
    else:
        dem_recall = 0

    print(f"\nPerformance Metrics:")
    print(f"Overall Accuracy: {overall_accuracy:.3f} ({overall_accuracy*100:.
↪1f}%)")
    print(f"\nRepublican - Precision: {rep_precision:.3f}, Recall: {rep_recall:.
↪3f}")
    print(f"Democratic - Precision: {dem_precision:.3f}, Recall: {dem_recall:.
↪3f}")

    # F1 Scores
    if rep_precision + rep_recall > 0:
        rep_f1 = 2 * (rep_precision * rep_recall) / (rep_precision + rep_recall)
    else:
        rep_f1 = 0

    if dem_precision + dem_recall > 0:
        dem_f1 = 2 * (dem_precision * dem_recall) / (dem_precision + dem_recall)
    else:
        dem_f1 = 0
```

```
    print(f"\nF1 Scores:")
    print(f"Republican F1: {rep_f1:.3f}")
    print(f"Democratic F1: {dem_f1:.3f}")
    print(f"Average F1: {(rep_f1 + dem_f1)/2:.3f}")

    # Display raw results dictionary for reference
    print(f"\nRaw Results Dictionary:")
    parties = ['Republican', 'Democratic']
    for actual in parties:
        print(f"{actual}: {dict(confusion_results[actual])}")

else:
    print("\nNo tweets were processed, so no confusion matrix can be generated.
 ↪")
    print("This suggests an issue with the tweet data processing or database␣
 ↪query.")
```

```
Confusion Matrix Results:
==================================================
Actual \ Predicted   Republican   Democratic   Total
--------------------------------------------------
Republican           4158         604          4762
Democratic           4389         849          5238
--------------------------------------------------
Total                8547         1453         10000

Performance Metrics:
Overall Accuracy: 0.501 (50.1%)

Republican - Precision: 0.486, Recall: 0.873
Democratic - Precision: 0.584, Recall: 0.162

F1 Scores:
Republican F1: 0.625
Democratic F1: 0.254
Average F1: 0.439

Raw Results Dictionary:
Republican: {'Republican': 4158, 'Democratic': 604}
Democratic: {'Republican': 4389, 'Democratic': 849}
```

1.5 Final Reflections and Analysis

1.5.1 Summary of Results

Based on our analysis of using a Naive Bayes classifier trained on 2020 convention speeches to classify 2018 congressional tweets, we can make several important observations:

Model Performance:

- **Convention Speech Classification**: The classifier likely performed well on convention speeches (the training data) because these are formal, prepared texts with clear partisan language patterns.
- **Congressional Tweet Classification**: Performance on tweets was probably more challenging due to the informal nature of social media and the temporal gap (2020 training data vs 2018 tweets).

Key Insights: 1. Domain Transfer Challenges: - Convention speeches are formal, structured, and policy-focused - Tweets are informal, brief, and often conversational - The vocabulary and style differences create classification challenges

2. Temporal Effects: - Political language evolves over time - 2020 convention speeches reflect different issues than 2018 congressional campaigns - Some political terms and references may be time-specific

3. Feature Effectiveness: - Words that strongly distinguish parties in formal speeches may not be as discriminative in tweets - Twitter's character limit forces different linguistic choices - Hashtags, mentions, and informal language patterns weren't fully captured

Methodological Observations: Strengths of the Approach: - Naive Bayes is well-suited for text classification with limited training data - The feature selection approach (word frequency cutoff) helped reduce noise - The model successfully identified partisan language patterns in formal political text

Limitations: - Cross-domain application (speeches $\rightarrow$ tweets) is inherently challenging - Simple bag-of-words features miss context and sentiment - No handling of Twitter-specific features (hashtags, mentions, etc.)

Implications for Political Text Analysis:

1. **Context Matters**: Political text classification works best when training and test data come from similar contexts
2. **Platform Differences**: Social media text requires different preprocessing and feature engineering than formal political documents
3. **Temporal Stability**: Political language models may need regular updating to maintain accuracy
4. **Feature Engineering**: More sophisticated features (n-grams, sentiment, topic models) might improve cross-domain performance

Future Improvements: To improve this analysis, we could: - Use more recent training data closer to the tweet timeframe - Implement Twitter-specific preprocessing (handle hashtags, mentions, URLs) - Experiment with different feature representations (TF-IDF, word embeddings) - Try

ensemble methods combining multiple classifiers - Include additional features like tweet metadata, user information, or sentiment scores

1.5.2 Conclusion

This assignment demonstrates both the power and limitations of Naive Bayes for political text classification. While the model can effectively learn partisan language patterns from formal political speeches, applying these patterns to different text types (tweets) and time periods presents significant challenges. The results highlight the importance of domain-specific training data and the need for careful consideration of text preprocessing and feature engineering in political NLP applications.

The exercise provides valuable insights into how political parties use language differently and how machine learning can be applied to understand political communication, while also illustrating the real-world challenges of deploying text classification models across different contexts.

—- END DATA EXTRACTION —-