# Political Naive Bayes

September 30, 2025

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

[81]:
```
import sqlite3
import nltk
import random
import pandas as pd
import numpy as np
import re
from collections import Counter, defaultdict

from string import punctuation

# Feel free to include your text patterns functions
# from text_functions_solutions import clean_tokenize, get_patterns
from functools import partial
from pathlib import Path
from nltk import sent_tokenize
from nltk.corpus import stopwords
```

**Text Preprocessing Pipeline Setup** The following are the helper functions for the text preprocessing pipeline. The pipeline includes the following transformations:

1. **Tokenize on whitespace** – split sentences into tokens based on spaces.
2. **Remove punctuation** – strip out punctuation marks from tokens.
3. **Keep only alphabetic tokens** – discard numbers, symbols, and mixed tokens.
4. **Remove stopwords** – filter out common words (e.g., "the", "and") that carry little meaning.
5. **Lowercase conversion** – normalize all words to lowercase.
6. **Join back to string** – reconstruct the cleaned tokens into a single string.

The `run_pipeline()` function executes these steps in sequence for any given sentence.

[82]:
```
# set data location
data_location = Path("../datasets")

# constants
PUNCT_SET = set(punctuation)
```

```
TW_PUNCT_SET = PUNCT_SET - {"#", "@"}
SW_ENG = stopwords.words("english")
WHITESPACE_RE = re.compile(r"\s+")
TWEET_RE = re.compile("(^b['\"])(.*)")


# helper functions
def tokenize_on_ws(text):
    return WHITESPACE_RE.split(text)


def remove_punct(tokens, punct_set=PUNCT_SET):
    cleaned = []
    for token in tokens:
        tok = "".join([ch for ch in token if ch not in punct_set])
        if tok:
            cleaned.append(tok)
    return cleaned


def is_alpha(tokens):
    return [token for token in tokens if token.isalpha()]


def remove_stopwords(tokens, sw=SW_ENG):
    tokens = [token for token in tokens if token.lower() not in sw]
    return tokens


def lowercase(tokens):
    return [token.lower() for token in tokens]


def join_to_string(tokens):
    return " ".join(tokens)


def clean_tweet(text):
    return TWEET_RE.match(text).group(2)


def run_pipeline(text, pipeline):
    tokens = str(text)

    for transform in pipeline:
        tokens = transform(tokens)
```

```
    return tokens
```

[83]:
```
convention_db = sqlite3.connect(data_location / "2020_Conventions.db")
convention_cur = convention_db.cursor()
```

## 1.1 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" exercise. First, we'll pull in the text for each party and prepare it for use in Naive Bayes.

[84]:
```
convention_data = []

# fill the above list up with items that are themselves lists. The
# sublists will have two elements. The first element in the sublist
# should be the speech in a single string. The second element
# of the sublist should be the party.

query_results = convention_cur.execute(
    """
    SELECT text, party
    FROM conventions
    WHERE party != 'Other';
    """
)

for row in query_results:
    # store the results in convention_data
    convention_data.append([row[0], row[1]])
```

[85]:
```
# it's a best practice to close up your DB connection when you're done
convention_db.close()
```

### Value counts of Democractic vs Republican Speeches

[86]:
```
party_counts = Counter(row[1] for row in convention_data)

total = party_counts["Democratic"] + party_counts["Republican"]
dem = party_counts["Democratic"] / total
rep = party_counts["Republican"] / total
print(f'Democratic Speeches: {party_counts["Democratic"]} ({dem * 100:.2f}%)')
print(f'Rebuplican Speeches: {party_counts["Republican"]} ({rep * 100:.2f}%)')
print(total)
```

```
Democratic Speeches: 1551 (61.04%)
Rebuplican Speeches: 990 (38.96%)
2541
```

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

[87]:
```
random.choices(convention_data, k=5)
```

[87]:
```
[['Indiana.', 'Republican'],
 ['We've brought together voices from every part of America.', 'Democratic'],
 ['Joe has always cared about military families. They've been through so much.
When I went to Iraq, one of the generals said, " I want to share the story with
you." In his daughter's class, it was a Christmas program and they were playing
the Ave Maria. And one of the little girls burst into tears and the teacher ran
over and said, "What's the matter? What's the matter?" And she said, "That's the
song they played at my daddy's funeral. He died in the war." The teacher had no
idea that that little girl's father had fought in the war and had died. And that
night I said to my staff, I'm a teacher, we can do better. We've got to do
better to help our military kids.',
  'Democratic'],
 ['The Bidens have a track record of helping military families. And we've seen
it with their work that they've done with joining forces, and how they were able
to rally a country behind us.',
  'Democratic'],
 ['Joe Biden wants to build an economy far better suited to our changing world.
Better for young people. Better for families working and raising their kids.
Better for people who lost jobs and need new ones. Better for farmers tired of
being collateral damage in trade wars. Better for workers caring for the sick,
elderly, and people with disabilities. Better because of a living wage and
access to affordable higher education and healthcare, including prescription
drugs, and to childcare, a secure retirement, and for the first time, paid
family and medical leave.',
  'Democratic']]
```

It'll be useful for us to have a large sample size than 2020 affords, since those speeches tend to be long and contiguous. Let's make a new list-of-lists called `conv_sent_data`. Instead of each first entry in the sublists being an entire speech, make each first entry just a sentence from the speech. Feel free to use NLTK's `sent_tokenize` function.

[88]:
```
conv_sent_data = []

nltk.download("punkt")
nltk.download("punkt_tab")
for speech, party in convention_data:
    sentences = sent_tokenize(speech)
    conv_sent_data.append([sentences[0], party])
```

```
[nltk_data] Downloading package punkt to /home/junc/nltk_data…
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package punkt_tab to /home/junc/nltk_data…
[nltk_data]   Package punkt_tab is already up-to-date!
```

Again, let's look at some random entries.

[89]:
```
random.choices(conv_sent_data, k=5)
```

[89]:
```
[['The great John Lewis would often quote the old African proverb, "When pray,
move your feet," and then challenge us to do just that.',
  'Democratic'],
 ['He brought together law enforcement, prosecutors, advocate, and survivors.',
  'Democratic'],
 ['Refusing to be told who makes decisions about her body or anyone else's.',
  'Democratic'],
 ['We are a people in a quandary about the present.', 'Democratic'],
 ['State or sovereignty.', 'Republican']]
```

Now it's time for our final cleaning before modeling. Go through `conv_sent_data` and take the following steps:

1. Tokenize on whitespace
2. Remove punctuation
3. Remove tokens that fail the `isalpha` test
4. Remove stopwords
5. Casefold to lowercase
6. Join the remaining tokens into a string

[90]:
```
# setup pipeline
speech_pipeline = [
    tokenize_on_ws,
    remove_punct,
    is_alpha,
    remove_stopwords,
    lowercase,
    join_to_string,
]

clean_conv_sent_data = []  # list of tuples (sentence, party), with sentence␣
 ↪cleaned

for idx, sent_party in enumerate(conv_sent_data):
    cleaned = run_pipeline(sent_party[0], speech_pipeline)
    if cleaned:
        clean_conv_sent_data.append((cleaned, sent_party[1]))

random.choices(clean_conv_sent_data, k=5)
```

[90]:
```
[('time recognize childcare part basic infrastructure nation', 'Democratic'),
 ('washington dc welcome republican national convention', 'Republican'),
 ('name stacia brightmon', 'Republican'),
 ('madeline lauf founder begin health nutritional company', 'Republican'),
 ('missouri', 'Republican')]
```

If that looks good, let's make our function to turn these into features. First we need to build our list of candidate words. I started my exploration at a cutoff of 5.

[91]:
```
word_cutoff = 5

tokens = [w for t, p in clean_conv_sent_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 628 as features in the model.
```

[92]:
```
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()
    for word in words:
        if word in fw:
            ret_dict[word] = True
```

```
    return ret_dict
```

[93]:
```
assert len(feature_words) > 0
assert conv_features("obama was the president", feature_words) == {
    "obama": True,
    "president": True,
}
assert conv_features("some people in america are citizens", feature_words) == {
    "people": True,
    "america": True,
    "citizens": 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.

[94]:
```
featuresets = [
    (conv_features(text, feature_words), party) for (text, party) in␣
 ↪convention_data
]
```

[95]:
```
random.seed(20220507)
random.shuffle(featuresets)

test_size = 500
```

[96]:
```
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
```

[97]:
```
classifier.show_most_informative_features(25)
```

```
Most Informative Features
             enforcement = True           Republ : Democr =     27.5 : 1.0
                   votes = True           Democr : Republ =     21.6 : 1.0
                 climate = True           Democr : Republ =     17.3 : 1.0
                   media = True           Republ : Democr =     15.9 : 1.0
              appreciate = True           Republ : Democr =     14.0 : 1.0
                    drug = True           Republ : Democr =     10.3 : 1.0
                 special = True           Republ : Democr =     10.3 : 1.0
                   local = True           Republ : Democr =      9.9 : 1.0
                   elect = True           Democr : Republ =      9.6 : 1.0
                    land = True           Republ : Democr =      8.9 : 1.0
                  cities = True           Republ : Democr =      8.4 : 1.0
                citizens = True           Republ : Democr =      8.4 : 1.0
                    flag = True           Republ : Democr =      8.4 : 1.0
                greatest = True           Republ : Democr =      8.4 : 1.0
```

```
          clean = True           Democr : Republ =      7.9 : 1.0
       freedom = True            Republ : Democr =      7.8 : 1.0
           law = True            Republ : Democr =      7.8 : 1.0
        senior = True            Republ : Democr =      7.8 : 1.0
        record = True            Republ : Democr =      7.3 : 1.0
      officers = True            Republ : Democr =      7.2 : 1.0
      grateful = True            Republ : Democr =      6.9 : 1.0
          grew = True            Republ : Democr =      6.9 : 1.0
        heroes = True            Republ : Democr =      6.8 : 1.0
        border = True            Republ : Democr =      6.7 : 1.0
          race = True            Republ : Democr =      6.7 : 1.0
```

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

### 1.1.1 My Observations

Out of all the most informative feature words, 21 of the 25 words are dominated by the Republican party. This is an interesting point because the original dataset was dominantly speeches made by Democrats with 61% and only 39% for Republican speeches. This shows that speech patterns by Republicans are very structured and use the same words. They are very aligned within their messaging. In contrast, the Democratic speeches are not as aligned and therefore the parties messaging and focus may not be as clear.

## 1.2 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.

[98]:
```
cong_db = sqlite3.connect(data_location / "congressional_data.db")
cong_cur = cong_db.cursor()
```

[99]:
```
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
```

[100]:
```
tweet_data = []

twitter_pipeline = [
    clean_tweet,
    tokenize_on_ws,
    partial(remove_punct, punct_set=TW_PUNCT_SET),
    is_alpha,
    lowercase,
    join_to_string,
]

# 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 result in results:
    cleaned = run_pipeline(result[2], twitter_pipeline)
    if cleaned:
        tweet_data.append((cleaned, result[1]))
```

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...

[101]:
```
random.seed(20201014)

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

**Value counts of Democratic vs Republican Tweets**

[102]:
```
twparty_counts = Counter(row[1] for row in tweet_data)

total = twparty_counts["Democratic"] + twparty_counts["Republican"]
dem = twparty_counts["Democratic"] / total
rep = twparty_counts["Republican"] / total
print(f"Democratic Tweets: {twparty_counts['Democratic']} ({dem * 100:.2f}%)")
print(f"Republican Tweets: {twparty_counts['Republican']} ({rep * 100:.2f}%)")
```

```
Democratic Tweets: 710814 (58.91%)
Republican Tweets: 495800 (41.09%)
```

[103]:
```
for tweet, party in tweet_data_sample:
    features = conv_features(tweet, feature_words)
    estimated_party = classifier.classify(features)

    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: rt and got you covered registration candidate how
Actual party is Democratic and our classifer says Democratic.

Here's our (cleaned) tweet: follow us on twitter like us on facebook too
```

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

Here's our (cleaned) tweet: to my jewish friends and family around the world
have an easy fast may you be sealed in the book of life gmar hatima tova and
dont forget to say a quick prayer for our fellows in the whose minds may not be
entirely on tonight gd bless
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: rt congressional candidate abigail spanberger had a
brilliant comeback to her gop opponent constantly calling her nancy
Actual party is Democratic and our classifer says Democratic.

Here's our (cleaned) tweet: thank you and yes only what is right for the
american people
Actual party is Republican and our classifer says Republican.

Here's our (cleaned) tweet: i think whats hes trying to say is he thinks i did a
good job on cnn hes just playing hard to get
Actual party is Democratic and our classifer says Republican.

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

Here's our (cleaned) tweet: rt just two weeks ago ted cruz celebrated gun
violence with the nra what should texas do with him ask
Actual party is Democratic and our classifer says Democratic.

Here's our (cleaned) tweet: trumps executive orders are rooted in an unamerican
hostility toward immigrant
Actual party is Democratic and our classifer says Democratic.

Here's our (cleaned) tweet: hambardzoum avagyannkhachatour avagyannhovsep
sarkissiannkhatchadour jingiriannalex petrosyannsarkis
Actual party is Democratic and our classifer says Democratic.
```

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

[104]:
```python
# 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
    # Now do the same thing as above, but we store the results rather
    # than printing them.
    features = conv_features(tweet, feature_words)
    # get the estimated party
    estimated_party = classifier.classify(features)
    # estimated_party = "Gotta fill this in"

    results[party][estimated_party] += 1

    if idx > num_to_score:
        break
```

### Confusion Matrix

[105]:
```
tp_rep = results["Republican"]["Republican"]
fn_rep = results["Republican"]["Democratic"]
fp_rep = results["Democratic"]["Republican"]
tn_rep = results["Democratic"]["Democratic"]

accuracy = (tp_rep + tn_rep) / (tp_rep + fn_rep + fp_rep + tn_rep)
precision = tp_rep / (tp_rep + fp_rep)
recall = tp_rep / (tp_rep + fn_rep)
f1 = (2 * precision * recall) / (precision + recall)

cm = pd.DataFrame(
    [[tp_rep, fn_rep], [fp_rep, tn_rep]],
    index=["Actual Rep", "Act Dem"],
    columns=["Pred Rep", "Pred Dem"],
)

print(cm)
print(f"\nAccuracy: {accuracy * 100:.1f}%")
print(f"Precision: {precision * 100:.1f}%")
print(f"Recall: {recall * 100:.1f}%")
print(f"F1: {f1 * 100:.1f}%")
```

```
            Pred Rep  Pred Dem
Actual Rep      2965      1074
Act Dem         4559      1404

Accuracy: 43.7%
Precision: 39.4%
Recall: 73.4%
F1: 51.3%
```

### 1.2.1 Reflections

There is a huge imbalance between the targets with 59% Democratic tweets vs 41% Republican tweets. Yet, the model is able to accurately predict the tweet as Republican about two times more than Democratic tweets. With that said, the accuracy of the model is less than random chance at 44%. The precision of the models shows a lot of false positives for the target class of "Republican". The recall tells us that the model is good at identifying the target class but also means a lot of misclassifying "Democrats".

Overall, the model is over-predicting Republican. As in the speeches, Republican speeches use more targeted words while Democratic speeches use more diverse language. Even with the class imbalance, the feature words in Republican speeches, and tweets for that matter, overshadow the feature words of Democratic speeches and tweets.