ArdernClaire_HW4

October 12, 2023

1 Naive Bayes for Sentiment Analysis

This assignment is comprised of two parts:

1. **Theory**: Solve the Naive Bayes exercises 4.1 and 4.2 from Chapter 4 in the J&M textbook. Reformulate NB to emphasize title words.
2. **Implementation**: You will implement and experiment with various feature engineering techniques in the context of Naive Bayes models for Sentiment classification of movie reviews.

We will use the NB model implemented in sklearn:

https://scikit-learn.org/stable/modules/naive_bayes.html

1.1 Write Your Name Here: Claire Ardern

2 Submission Instructions

1. Click the Save button at the top of the Jupyter Notebook.
2. Please make sure to have entered your name above.
3. Select Cell -> All Output -> Clear. This will clear all the outputs from all cells (but will keep the content of all cells).
4. Select Cell -> Run All. This will run all the cells in order, and will take several minutes.
5. Once you've rerun everything, select File -> Download as -> PDF via LaTeX and download a PDF version *SentimentAnalysis.pdf* showing the code and the output of all cells, and save it in the same folder that contains the notebook file *SentimentAnalysis.ipynb*.
6. Look at the PDF file and make sure all your solutions are there, displayed correctly. The PDF is the only thing we will see when grading!
7. Submit **both** your PDF and notebook on Canvas.
8. Verify your Canvas submission contains the correct files by downloading it after posting it on Canvas.

Make sure that you format your solutions to theory questions to show equations properly. We will not grade solutions that are not properly formatted. Jupyter-notebook understands Latex, alternatively you can edit in Word using its Equation editor and submit the PDF as a separate file in Canvas.

3 Theory

3.1 Theory: J&M Exercise 4.1

Solve for probability of the sentence belonging to each of the two classes.

First, the positive class:

$$P(pos|s) = P(I|pos) + P(always|pos) + P(like|pos) + P(foreign|pos) + P(films|pos)$$

$$= (0.09)^1 \times (0.07)^1 \times (0.29)^1 \times (0.04)^1 \times (0.08)^1$$

$$P(pos|s) = 5.8464 \times 10^{-6}$$

Now, the negative class:

$$P(neg|s) = P(I|neg) + P(always|neg) + P(like|neg) + P(foreign|neg) + P(films|neg)$$

$$= (0.16)^1 \times (0.06)^1 \times (0.06)^1 \times (0.15)^1 \times (0.11)^1$$

$$P(neg|s) = 0.54$$

The greater probability of P(neg|s) clearly indicates that the sentence belongs to the negative class.

3.2 Theory: J&M Exercise 4.2

First, we must compute all of the probabilities for each class.

For the Comedy class:

$$P(c) = \frac{2}{5}$$

$$P(fun|c) = \frac{3}{9+7}$$

$$P(couple|c) = \frac{2}{9+7}$$

$$P(love|c) = \frac{2}{9+7}$$

$$P(fast|c) = \frac{1}{9+7}$$

$$P(furious|c) = \frac{0}{9+7}$$

$$P(shoot|c) = \frac{0}{9+7}$$

$$P(fly|c) = \frac{1}{9+7}$$

For the Action class:

$$P(fun|a) = \frac{1}{11+7}$$

$$P(couple|a) = \frac{0}{11+7}$$
$$P(love|a) = \frac{1}{11+7}$$
$$P(fast|a) = \frac{2}{11+7}$$
$$P(furious|a) = \frac{2}{11+7}$$
$$P(shoot|a) = \frac{4}{11+7}$$
$$P(fly|a) = \frac{1}{11+7}$$

We have cases of zero probabilities in each class! We must use Laplace Smoothing to resolve this issue. We are left with these new probabilities:

For the Comedy class:

$$P(c) = \frac{2}{5}$$
$$P(fun|c) = \frac{4}{16}$$
$$P(couple|c) = \frac{3}{16}$$
$$P(love|c) = \frac{3}{16}$$
$$P(fast|c) = \frac{2}{16}$$
$$P(furious|c) = \frac{1}{16}$$
$$P(shoot|c) = \frac{1}{16}$$
$$P(fly|c) = \frac{2}{16}$$

For the Action class:

$$P(fun|a) = \frac{2}{18}$$
$$P(couple|a) = \frac{1}{18}$$
$$P(love|a) = \frac{2}{18}$$
$$P(fast|a) = \frac{3}{18}$$
$$P(furious|a) = \frac{3}{18}$$
$$P(shoot|a) = \frac{5}{18}$$

$$P(fly|a) = \frac{2}{18}$$

Now, using these probabilities, we can compute which class the new document D belongs to. First, lets calculate the probability for the action class:

$$P(a|D) = P(fast|a) + P(couple|a) + P(shoot|a) + P(fly|a)$$

$$= (\frac{3}{19})^1 \times (\frac{1}{19})^1 \times (\frac{5}{19})^1 \times (\frac{2}{19})^1$$

$$P(a|D) = 2.3 \times 10^{-4}$$

Now, lets calculate the probability for the comedy class:

$$P(c|D) = P(fast|c) + P(couple|c) + P(shoot|c) + P(fly|c)$$

$$= (\frac{2}{17})^1 \times (\frac{3}{17})^1 \times (\frac{1}{17})^1 \times (\frac{2}{17})^1$$

$$P(c|D) = 1.4 \times 10^{-4}$$

As the probability of P(a|D) is greater, we can say the new document D belongs to the action class.

3.3 Theory 5111: Title is K times more important than Body

Mandatory for graduate students, optional for undergraduate students

The Naive Bayes algorithm for text categorization presented in class treats all sections of a document equally, ignoring the fact that words in the title are often more important than words in the text in determining the document category. Modify the Naive Bayes training algorithm to reflect that word occurrences in the title are K times more important than word occurrences in the rest of the document for deciding the class, where K is an input parameter. Describe the idea in English and include pseudocode, akin to the training pseudocode shown in class.

3.4 Solution:

In order to modify the Naive Bayes training algorithm to reflect that word occurrences in the title are K times more important than word occurrences in the rest of the document for deciding the class, we must add weight to the title words so that they hold K times more weight. This can be done when counting the occurrences of words in a document. For title words, we can add K times more weight by adding K word count for each one occurrence of the title word. The rest of the words in the document (not title words) would continue to add one word count for each one occurrence of the word. Pseudocode for this idea is shown below:

for each category C_k: let D_k be the subset of documents in category C_k set $p(C_k) = |D_k|/|D|$ let t_k be the total number of title words in D_k let n_k be the total number of body words in D_k for each word w_i in V: let n_{ki} by the number of occurrences of w_i in the body of D_k let t_{ki} by the number of occorrences of w_i in the title of D_k set $p(w_i|C_k) = (n_{ki} + K * t_{ki} + 1)/(n_k + t_k + |V|)$

4 Implementation

4.1 From documents to feature vectors

This section illustratess the prototypical components of machine learning pipeline for an NLP task, in this case document classification:

1. Read document examples (train, devel, test) from files with a predefined format:
   - assume one document per line, usign the format "<label> <text>".
2. Tokenize each document:
   - using a spaCy tokenizer.
3. Feature extractors:
   - so far, just words.
4. Process each document into a feature vector:
   - map document to a dictionary of feature names.
   - map feature names to unique feature IDs.
   - each document is a feature vector, where each feature ID is mapped to a feature value (e.g. word occurences).

[1]:
```python
import spacy
from spacy.lang.en import English
from scipy import sparse
from sklearn.naive_bayes import MultinomialNB
```

[2]:
```python
# Create spaCy tokenizer.
spacy_nlp = English()

def spacy_tokenizer(text):
    tokens = spacy_nlp.tokenizer(text)

    return [token.text for token in tokens]
```

[3]:
```python
def read_examples(filename):
    X = []
    Y = []
    with open(filename, mode = 'r', encoding = 'utf-8') as file:
        for line in file:
            [label, text] = line.rstrip().split(' ', maxsplit = 1)
            X.append(text)
            Y.append(label)
    return X, Y
```

[4]:
```python
def word_features(tokens):
    feats = {}
    for word in tokens:
        feat = 'WORD_%s' % word
        if feat in feats:
            feats[feat] +=1
        else:
            feats[feat] = 1
    return feats
```

[5]:
```
def add_features(feats, new_feats):
    for feat in new_feats:
        if feat in feats:
            feats[feat] += new_feats[feat]
        else:
            feats[feat] = new_feats[feat]
    return feats
```

This function tokenizes the document, runs all the feature extractors on it and assembles the extracted features into a dictionary mapping feature names to feature values. It is important that feature names do not conflict with each other, i.e. different features should have different names. Each document will have its own dictionary of features and their values.

[6]:
```
def docs2features(trainX, feature_functions, tokenizer):
    examples = []
    count = 0
    for doc in trainX:
        feats = {}

        tokens = tokenizer(doc)

        for func in feature_functions:
            add_features(feats, func(tokens))

        examples.append(feats)
        count +=1

        if count % 100 == 0:
            print('Processed %d examples into features' % len(examples))

    return examples
```

[7]:
```
# This helper function converts feature names to unique numerical IDs.

def create_vocab(examples):
    feature_vocab = {}
    idx = 0
    for example in examples:
        for feat in example:
            if feat not in feature_vocab:
                feature_vocab[feat] = idx
                idx += 1

    return feature_vocab
```

[8]:
```
# This helper function converts a set of examples from a dictionary of feature␣
 ↪names to values representation
# to a sparse representation of feature ids to values. This is important␣
 ↪because almost all feature values will
# be 0 for most documents and it would be wasteful to save all in memory.

def features_to_ids(examples, feature_vocab):
    new_examples = sparse.lil_matrix((len(examples), len(feature_vocab)))
    for idx, example in enumerate(examples):
        for feat in example:
            if feat in feature_vocab:
                new_examples[idx, feature_vocab[feat]] = example[feat]

    return new_examples
```

[9]:
```
# Evaluation pipeline for the Naive Bayes classifier.

def train_and_test(trainX, trainY, devX, devY, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))
```

[10]:
```
import os

datapath = '../data'
```

```python
train_file = os.path.join(datapath, 'imdb_sentiment_train.txt')
trainX, trainY = read_examples(train_file)

dev_file = os.path.join(datapath, 'imdb_sentiment_dev.txt')
devX, devY = read_examples(dev_file)

# Specify features to use.
features = [word_features]

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 28692
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.779
```

4.2 Feature engineering

Evaluate NB model performance when using only alpha tokens. This can be done by changing the tokenizer function.

[11]:
```python
def spacy_tokenizer1(text):
    tokens = spacy_nlp.tokenizer(text)

    # YOUR CODE HERE
    # Keep in the tokens list only those whose text is made up solely from␣
 ↪letters.
    tokens = [token.text for token in tokens if token.text.isalpha()]

    return tokens

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer1)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 25054
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.785
```

Same as above, but lowercase all tokens before using as features.

```
[12]: def spacy_tokenizer2(text):
          tokens = spacy_nlp.tokenizer(text)

          # YOUR CODE HERE
          # Keep in the tokens list only those whose text is made up solely from␣
       ↪letters.
          # Return a list of lowercased token text.
          tokens = [token.text.lower() for token in tokens if token.text.isalpha()]

          return tokens

      # Evaluate NB model.
      train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 21708
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.784
```

Same as above, but lowercase only tokens that appear at the beginning of sentences.

```
[13]: spacy_nlp = English()
spacy_nlp.add_pipe("sentencizer")

def spacy_tokenizer3(text):
    doc = spacy_nlp(text)

    text_new = ''

    for sent in doc.sents:
        #print(sent)
        if sent[0].text.isalpha():
            first = sent[0].text.lower()
            rest = sent[1:].text
        else:
            first = sent[0].text
            rest = sent[1:].text

        #print(first + " " + rest)
        text_new = text_new + " " + first + " " + rest

    tokens = spacy_nlp.tokenizer(text_new)

    # YOUR CODE HERE

    return tokens

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer3)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 28707
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.779
```

Use spacy_tokenizer2 (only alpha tokens, lowered all) and display the top 10 most frequent tokens in the vocabulary, as a list of tuples (token, frequency).

[14]:
```python
# First, count token occurrences across all examples, where features are still␣
↪strings.
def create_feature_counts(examples):
    feature_counts = {}

    for features in examples:
        for feature in features:
            feature_counts[feature] = feature_counts.get(feature, 0) + 1

    return feature_counts

# Create features for all training examples, compute feature counts
def fcounts_from_train(trainX, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_counts = create_feature_counts(trainX_feat)
    print('Vocabulary size: %d' % len(feature_counts))

    return feature_counts

# Return a list of the top K most frequent tokens in the vocabulary.
def topK_tokens(vocab, k):
    # YOUR CODE HERE
    ranked = [(k, v) for k, v in vocab.items()]
    ranked.sort(key=lambda x: x[1], reverse=True)

    return ranked[:k]

vocab = fcounts_from_train(trainX, features, spacy_tokenizer2)
# Instructions say to return the first 10, but provided code returns the first␣
 ↪20.
stop_words = topK_tokens(vocab, 20)
for item in stop_words:
    print(item)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 21708
('WORD_the', 1487)
('WORD_a', 1461)
('WORD_and', 1459)
('WORD_of', 1418)
('WORD_to', 1417)
('WORD_is', 1349)
('WORD_this', 1343)
('WORD_in', 1313)
('WORD_it', 1311)
('WORD_that', 1209)
('WORD_i', 1168)
('WORD_for', 1076)
('WORD_but', 1064)
('WORD_with', 1061)
('WORD_was', 986)
('WORD_as', 960)
('WORD_on', 944)
('WORD_not', 907)
('WORD_are', 884)
```

```
('WORD_movie', 878)
```

Evaluate NB model performance when ignoring the top 20 stop words. Use spacy_tokenizer2 (only alpha tokens, lowered all).

[15]:
```python
# Evaluation pipeline for the Naive Bayes classifier.
from collections import Counter

def train_and_test(trainX, trainY, devX, devY, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)
    #print(trainX_feat[1])

    # Create vocabulary from features in training examples.
    feature_counts = create_feature_counts(trainX_feat)
    stop_words = topK_tokens(vocab, 20)

    # Remove from each example features that appear in the stop words list.
    # YOUR CODE HERE.
    for doc in trainX_feat:
        for word in stop_words:
            if word[0] in doc.keys():
                del doc[word[0]]

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))

# Evaluate NB model.
train_and_test(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 21688
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.801
```

Evaluate NB model performance when ignoring words that appear less than 5 times. Use spacy_tokenizer2 (only alpha tokens, lowered all).

[16]:
```python
# Evaluation pipeline for the Naive Bayes classifier.

def train_and_test_2(trainX, trainY, devX, devY, feature_functions, tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_counts = create_feature_counts(trainX_feat)
    #print(feature_counts)

    # Remove from each example features that appear in the stop words list.
    # YOUR CODE HERE.
    for doc in trainX_feat:
        for feature in feature_counts.items():
#             print(feature)
#             print(feature[0])
#             print(feature[1])
            if feature[1] < 5:
                if feature[0] in doc.keys():
                    #print(doc[feature[0]])
                    del doc[feature[0]]

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))

# Evaluate NB model.
train_and_test_2(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 4872
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.796
```

4.3 Binary Multinomial Bayes

Mandatory for graduate students, optional for undergraduate students

Write code for transforming documents to features such that features are Boolean and only represent whether a word occurred in a document, as in the Binary Multinomial Naive Bayes discussed in class. Evaluate the Naive Bayes model with this feature representation, using spacy_tokenizer2 (only alpha tokens, lowered all).

[17]:
```python
def add_features_bool(feats, new_feats):

    for feat in new_feats:
        if feat not in feats:
            feats[feat] = True

    return feats

def docs2features_bool(trainX, feature_functions, tokenizer):

    examples = []
    count = 0
    for doc in trainX:
        feats = {}

        tokens = tokenizer(doc)

        for func in feature_functions:
            add_features_bool(feats, func(tokens))

        examples.append(feats)
        count +=1

        if count % 100 == 0:
            print('Processed %d examples into features' % len(examples))

    return examples
```

```python
def train_and_test_bool(trainX, trainY, devX, devY, feature_functions,␣
 ↪tokenizer):
    # Pre-process training documents.
    trainX_feat = docs2features_bool(trainX, feature_functions, tokenizer)

    # Create vocabulary from features in training examples.
    feature_vocab = create_vocab(trainX_feat)
    print('Vocabulary size: %d' % len(feature_vocab))

    trainX_ids = features_to_ids(trainX_feat, feature_vocab)

    # Train NB model.
    nb_model = MultinomialNB(alpha = 1.0)
    nb_model.fit(trainX_ids, trainY)

    # Pre-process test documents.
    devX_feat = docs2features_bool(devX, feature_functions, tokenizer)
    devX_ids = features_to_ids(devX_feat, feature_vocab)

    # Test NB model.
    print('Accuracy: %.3f' % nb_model.score(devX_ids, devY))

train_and_test_bool(trainX, trainY, devX, devY, features, spacy_tokenizer2)
```

```
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Vocabulary size: 21708
Processed 100 examples into features
Processed 200 examples into features
Processed 300 examples into features
Processed 400 examples into features
Processed 500 examples into features
Processed 600 examples into features
Processed 700 examples into features
Processed 800 examples into features
Processed 900 examples into features
Processed 1000 examples into features
Processed 1100 examples into features
Processed 1200 examples into features
Processed 1300 examples into features
Processed 1400 examples into features
Processed 1500 examples into features
Accuracy: 0.807
```

4.4 Bonus points

Anything extra goes here. For example, implement NB from scratch in a separate module nbayes.py and use it for the exercises above.

```
[ ]:
```

4.5 Analysis

Include an analysis of the results that you obtained in the experiments above. Take advantage of the Jupyter Notebook markdown language, which can also process Latex and HTML, to format your report so that it looks professional.

4.5.1 The Differences Between Our Models

One of the many ways to improve model performance is to provide more data for training. Since there are so many potential forms that an object of a class may take, especially in NLP, the more information the model sees, the easier it will be for the model to generalize and make accurate predicitons. In addition, the type or format of the information that is given to the model may influence how well it performs. This is because there are underlying patterns within the data that allow the model to make connections between features and classes. Some connections may be stronger or more obvious than others which can be noted in a comparison between resulting model performances when different features are used.

4.5.2 Feature Engineering Analysis: Part 1 (only alpha tokens)

With the first model experiment, we used only alpha tokens as the data passed to the model. In this case, we had a vocabulary size of 25,054. This means that the model has 25,054 words that can be used in the process of identifying the class of an object. With this vocabulary size, an accuracy of 0.785 was achieved. While this isn't necessarily a bad accuracy level, it could be much much better. By comparing with the performance of the other experiemnts, we can see how changes in the data/features given to the model may or may not improve upon this accuracy score.

4.5.3 Feature Engineering Analysis: Part 2 (only alpha tokens, all lowercase)

With the second model experiment, we lowercased all alpha tokens before passing them to the model. This resulted in a vocabulary size of 21,708, which is smaller than in the previous experiment. The vocabulary is smaller after lowercasing because any difference between words due to uppercase letters has been removed, consolodating many cases of different word representations into one word. So, although we technically have the same number of words, we have a smaller number of different word representations, which often hold different meanings. Some information may have been lost here.

This model experiment resulted in an accuracy of 0.784. This is only 0.1% lower than the previous experiment, so it is not a drastic difference, though we can still see this minor loss of information reflected in the model accuracy. Based on these results, it would be reasonable to conclude that lowercasing all tokens before passign them to the model is not beneficial to the model performance.

4.5.4 Feature Engineering Analysis: Part 3 (only alpha tokens, lowercase start of sentence)

With the third model experiment, we used only alpha tokens and lowercased only the tokens at the beginning of a sentence. This is an interesting modification of the previous experiment. By doing so, we had a vocabulary size of 28,707, which is larger than the vocabulary size in each of the previous experiments. While it is obvious that we lost some information by lowercasing all alpha tokens, it seems we may have gained some information by lowercasing only the alpha tokens at the beginning of a sentence. This may be beause the upper or lower case representations of a word may have different meanings, though all words at the beginning of a sentence must be upper case due to their position, not due to the word meaning. Removing ambiguous cases of uppercase words such as those at the beginning of a sentence could make the information provided to the model more clear.

This model experiment resulted in an accuracy of 0.779, which is lower than both of the previous experiments. This likely means that although we provided the model with more information in terms of size, we may have lost some information in terms of clarity, weight, or importance. As stated in the introduction to the analysis, there are underlying patterns and connections within the data and some of these patterns or connections may be stronger than others. It is clear based on this experiment and the previous experiment that lowercasing the alpha tokens alone removes important connections within the data that help to improve model accuracy. So, if we want to improve model performance, we must make changes to the features elsewhere.

4.5.5 Feature Engineering Analysis: Part 4 (removing the top 20 stop words)

With the fourth model experiment, we removed the top 20 most frequently occuring stop words. These are words such as "the," "and," "of," and so on. While these words are very common and will likely occur several times within a document, there is not much meaning that is carried by these words. Especially in the case of determining a class, since documents of different classes are all likely to contain several instances of these words, there is little to no information within those tokens that will assist the model in differentiating between classes. By removing these words, we were left with a vocabulary size of 21,688. This is not far off from the vocabulary size in Part 2, which is the vocabulary we started with before removing the stop words. This vocabulary size difference of 20 makes perfect sense as we have removed 20 words from the vocabulary here.

This model experiment reslted in an accuracy of 0.801, which is the highest accuracy we have seen across all of our experiemnts so far. The increase in accuracy is likely due to the fact that we have removed unnecessary information that really only serves to clutter and confuse. These extra stop words may distract from more important words that can be used to differentiate between classes. In addition, since these words occur so frequently, it is likely that results were being skewed one way or the other. So, we can see here that more data doesn't always mean better performance. The actual information that is held within the data is often more important.

4.5.6 Feature Engineering Analysis: Part 5 (ignore words with count less than 5)

With the fifth model experiment, we ignored all words with a count less than 5. This removes any words that have a particularly small probability in comparison with the rest of the words. By doing so, we may continue the process of removing extraneous information that was started in Part 4. When we do this, our vocabulary size drops to 4,872, which is much smaller than the vocabulary used in all of the previous experiments. Based on this change, it seems there are many words that are included in the vocabulary that do not appear in any class particularly frequently. It is possible that these words are also providing more confusion than clarity.

This model experiment resulted in an accuracy of 0.796 which is not the highest score, but is a close second. It seems that although the words that we removed from the vocabulary did not occur many times, they still had some relevent weight. Perhaps the fact that these words are so rare is helpful in differentiating between classes. If a word is seen very rarely, but only seen in one specific class, this can be very helpful to the model when learning the differences between classes. So, while removing these words was slighly helpful in comparison to Parts 1-3, it seems it is still better to proceed with the steps taken in Part 4.

4.5.7 Binary Multinomial Bayes

The final model experiment, an implementation of Binary Multinomial Bayes, is different from the experiments discussed above. In this case, we are removing word counts altogether. Instead, we are simply tracking whether or not the word appears in the document. This way, instead of comparing how many times you would see a word in a document of each class, we are comparing whether a word appears in a document of each class. By doing so, we have a vocabulary size of 21,708, which seems fairly average.

This model experiment resulted in an accuracy of 0.807, which is the highest of all of the experiments. It seems that the Binary Multinomial Bayes model performed better than the previous implementations of the Naive Bayes model. This could mean that it is easier to make a connection between features and classes rather than features, feature counts, and classes.

```
[ ]:
```