Homework 1: Sentiment Analysis with Naïve Bayes

CSCI 3832 Natural Language Processing

1. Lemmas and inflected forms, hyponyms/hypernyms, the distributional hypothesis
2. Tokenization, vocabularies, and feature extraction for a Naive Bayes model

Julia Troni

julia.troni@colorado.edu or jutr6738@colorado.edu

Section 1: Free Response Questions

Question 1: Write down the lemmas of the following inflected forms:

1. walked
2. taught
3. best
4. are
5. running

Your answer here

1. walk
2. teach
3. good
4. be
5. run

Question 2: Write down 3 hyponyms of the following words:

1. dog
2. food
3. profession

Your answer here

1. dog- golden retriever, lab, border collie
2. food - peanut butter, banana, tofu
3. profession- teacher, chef, doctor

Question 3: In your own words, describe:

1. The distributional hypothesis (see lecture on distributional semantics)
2. How is the distributional hypothesis relvant to NLP systems?

Your answer here

The distributional hypothesis states that words that occur in similar contexts tend to have similar meanings. This is relevant to NLP systems because it is the foundation for many techniques, such as word embeddings, which can then be used as input for many NLP applications such as language translation and text classification. Also,the distributional hypothesis can also be used to develop models of word meaning and similarity that can be used in tasks such as word sense disambiguation.

Section 2: Sentiment Analysis with Naive Bayes

In this section, our goal is to classify a set of movie reviews as positive or negative. For our dataset, we'll use the Large Movie Review Dataset. To get started, download the dataset from the link, and extract it to where your notebook is. Next, we'll load the data and look at a couple of examples.

*Important: for any project which involves creating or training models, you can **only** do your exploratory data analysis on the training set. Looking at the test set in any way can invalidate your results!*

In [10]:

```
import os

data_dir = 'aclImdb/'

pos_train_dir = data_dir + 'train/pos/'
neg_train_dir = data_dir + 'train/neg/'

def read_folder(folder):
    examples = []
    for fname in os.listdir(folder):
        with open(os.path.join(folder, fname), encoding='utf8') as f:
            examples.append(f.readline().strip())
    return examples

pos_examples = read_folder(pos_train_dir)
neg_examples = read_folder(neg_train_dir)

print('Number of positive examples: {}\nNumber of negative examples: {}\n\n'.format(len

print('Sample positive example: {}\n\n'.format(pos_examples[0]))
print('Sample negative example: {}'.format(neg_examples[0]))
```

```
Number of positive examples: 12500
Number of negative examples: 12500


Sample positive example: Bromwell High is a cartoon comedy. It ran at the same time as s
ome other programs about school life, such as "Teachers". My 35 years in the teaching pr
ofession lead me to believe that Bromwell High's satire is much closer to reality than i
s "Teachers". The scramble to survive financially, the insightful students who can see r
ight through their pathetic teachers' pomp, the pettiness of the whole situation, all re
mind me of the schools I knew and their students. When I saw the episode in which a stud
ent repeatedly tried to burn down the school, I immediately recalled ......... at
.......... High. A classic line: INSPECTOR: I'm here to sack one of your teachers. STUDE
NT: Welcome to Bromwell High. I expect that many adults of my age think that Bromwell Hi
gh is far fetched. What a pity that it isn't!
```

```
Sample negative example: Story of a man who has unnatural feelings for a pig. Starts out
with a opening scene that is a terrific example of absurd comedy. A formal orchestra aud
ience is turned into an insane, violent mob by the crazy chantings of it's singers. Unfo
rtunately it stays absurd the WHOLE time with no general narrative eventually making it
just too off putting. Even those from the era should be turned off. The cryptic dialogue
would make Shakespeare seem easy to a third grader. On a technical level it's better tha
n you might think with some good cinematography by future great Vilmos Zsigmond. Future
stars Sally Kirkland and Frederic Forrest can be seen briefly.
```

Now that we've loaded the data, let's create our vocabulary. While we want our vocabulary to cover the whole training set, we'll keep them separate to see if there are any words which are frequently found in one or the other class -- these words might be informative features for classification!

The simplest way to create a vocabulary is to split on spaces:

In [11]:

```python
pos_words = []  # A list of all space separated tokens found across all positive exampl
neg_words = []

pos_vocab = set()  # A list of *unique* separated tokens found in across all positive e
neg_vocab = set()
```

In [12]:

```python
##Counts of total words (tokens)

pos_words=[]  # A list of all space separated tokens found across all positive examples

for pos in pos_examples:
    strs=pos.split(' ')
    for s in strs:
        pos_words.append(s)

neg_words = []
for neg in neg_examples:
    strs=neg.split(' ')
    for s in strs:
        neg_words.append(s)

#counts of unique words (types)
pos_vocab = set(pos_words)  # A list of *unique* separated TYPES found in across all po
neg_vocab = set(neg_words)
```

In [13]:

```python
# Sanity check

print(len(pos_words))
print(len(pos_vocab))

assert len(pos_words) == 2958696
assert len(pos_vocab) == 178873
```

```
2958696
178873
```

Now lets calculate word frequencies for each class. (Hint: use the Python Counter class)

In [14]:

```python
pos_frequencies = [] # A list of tuples of the form (word, count).
```

```python
                      # The list should be sorted in descending order, using the count of ea

neg_frequencies = []
```

In [15]:

```python
from collections import Counter

'''
Your code here. For each class (positive/negative) calculate the frequency of each word
and neg_counter.

Print the top 15 most common word for each class.

'''

pos_frequencies = Counter(pos_words).most_common() # A list of tuples of the form (word
                      # The list should be sorted in descending order, using the count of ea

neg_frequencies = Counter(neg_words).most_common()
```

In [16]:

```python
assert pos_frequencies[0] == ('the', 148413)
assert neg_frequencies[0] == ('the', 138612)
```

In [17]:

```python
print("Top 15 positive words", pos_frequencies[0:15])
print('')
print("Top 15 negative words", neg_frequencies[0:15])
```

```
Top 15 positive words [('the', 148413), ('and', 84270), ('a', 79427), ('of', 75341), ('t
o', 65209), ('is', 55358), ('in', 45794), ('that', 31941), ('I', 30927), ('it', 26987),
('this', 26021), ('/><br', 24617), ('as', 23930), ('with', 22031), ('was', 21308)]

Top 15 negative words [('the', 138612), ('a', 75665), ('and', 68381), ('of', 67629), ('t
o', 67359), ('is', 47870), ('in', 39782), ('I', 35043), ('that', 32615), ('this', 3117
7), ('it', 27440), ('/><br', 26318), ('was', 25389), ('for', 20197), ('with', 19687)]
```

Looking at the top 15 words for each class we see two problems:

1. The words are essentially the same for each class, which doesn't give us any information on how to differentiate them.
2. Look at the most frequent tokens. Are there any tokens which aren't words? Any situations where tokens with different surface forms but the same meaning could be repeated (and if so, how might we control for this?)

Your answer to 2 here

- Tokens that appear that are not words but still represent meaning:
    - punctuation like "...", "<br", "br/>", "/>"
    - emoji representations such as XD, :p, =], O_O
    - numbers

- Tokens with different surface forms but the same meaning that are repeated:
    - Uppercase vs. lowercase
        - And vs. and

- OK vs. Ok vs. ok
- We can control for this by converting the list of words to all lowercase using .tolower()

- With vs. without punctuation and differences in spacing:
  - and...funny vs. and... vs. funny

Instead of looking at the most frequent words, let's instead look at the most frequent words which explicitly do not appear in the other class.

In [18]:

```
only_pos_words = [word for word in pos_words if word not in neg_vocab]
only_neg_words = [word for word in neg_words if word not in pos_vocab]

opw_counter = Counter(only_pos_words)
onw_counter = Counter(only_neg_words)

print(opw_counter.most_common()[:50])
print('\n')
print(onw_counter.most_common()[:50])
```

```
[('Edie', 82), ('Gundam', 74), ('Antwone', 58), ('/>8/10', 47), ('/>7/10', 46), ('/>10/1
0', 45), ('Gunga', 44), ('Gypo', 44), ('Din', 43), ('Othello', 41), ('7/10.', 37), ('Blu
nt', 37), ('Yokai', 37), ('Tsui', 35), ('Blandings', 34), ('Goldsworthy', 32), ('/>9/1
0', 31), ('Gino', 31), ('Visconti', 30), ('Bernsen', 29), ('Taker', 29), ('Brashear', 2
9), ('Harilal', 29), ('Clutter', 28), ("Goldsworthy's", 27), ('"Rob', 26), ('Dominick',
25), ('MJ', 25), ('/>7', 24), ('Rosenstrasse', 24), ('Sassy', 24), ('Flavia', 24), ('Ash
raf', 23), ('Recommended.', 22), ('Brock', 22), ('vulnerability', 22), ('Sabu', 22), ('K
orda', 22), ('Ahmad', 22), ('Stevenson', 22), ('Coop', 22), ('Riff', 22), ('flawless.',
21), ('aunts', 21), ("Gilliam's", 21), ('Solo', 21), ('Kells', 21), ("Capote's", 21),
('Cutter', 21), ('Blackie', 21)]


[('/>4/10', 56), ('/>Avoid', 55), ('2/10', 49), ('*1/2', 45), ('unwatchable.', 43), ('/>
3/10', 40), ('Thunderbirds', 40), ('Gamera', 39), ('steaming', 35), ('Wayans', 33), ('Sl
ater', 31), ('drivel.', 30), ('Tashan', 29), ('Aztec', 29), ('/>1/10', 28), ('Sarne', 2
7), ('Kareena', 26), ('BTK', 26), ('Segal', 26), ('blah,', 26), ('Delia', 26), ('0/10',
25), ('neither.', 25), ('Gram', 25), ('(*1/2)', 24), ('croc', 24), ('Dahmer', 24), ('Dar
kman', 24), ('Rosanna', 23), ('Zenia', 23), ('tripe.', 22), ('awful!', 22), ('2/10.', 2
2), ('Kornbluth', 22), ('Saif', 21), ('incoherent,', 21), ('appallingly', 21), ('Shaq',
21), ('Welch', 21), ('Hackenstein', 21), ('/>2/10', 20), ('4/10.', 20), ('kibbutz', 20),
('Clay', 20), ('Morgana', 20), ('"1"', 19), ('crawling', 19), ('/>1', 19), ('awfulness',
19), ('Mraovich', 19)]
```

We begin to see some words we would expect to denote a negative review, but not so much for the positive reviews. Why might this be the case? What types of tokens are found in positive reviews but not in negative reviews?

In [ ]:

Your answer here

The negative reivew words contain words such as "unwachable", "appalling", "incoherent" which are very indicative of negative sentiment. Positive review words contain far more proper nouns and names. This may be because positive reviews seem to give more of a description and summary of plot, rather than negative reviews are very to the point and explicit that the movie was bad.

In [ ]:

```

```

In [19]:

```
# Lets now make our combined vocabulary
space_vocab = list(pos_vocab.union(neg_vocab))
print('Length of space separated vocab: {}'.format(len(space_vocab)))
print(space_vocab[:50])
```

```
Length of space separated vocab: 281137
['b****es!', 'Lacan),', 'Tibbett.', 'memory-erasing', 'Central.', 'Playmates.I', '"Offsi
de".', 'naivete', 'Yorker', 'movers', 'together?).', 'performance.Tim', 'favorably.', 'F
at/Andy', 'Iliad', 'hat-check', 'worst!).', 'cult-members', 'feel...well,', 'Chance."<b
r', 'Gateshead', 'CASPER', 'ear-pleasing', ',most', 'eyeballs.', 'bravery,', '/>Thurma
n', 'everytime.', "cover'", 'sir,', 'Yum', 'headquartered', 'MEN)', 'Coolio', 'Katsumi',
'well-structured', 'sympathy.', 'Knightley', 'all-time!', "Borzage's", "Seeber's", 'Pros
titute(which', 'cut.', 'Relentlessy', 'Independence)', 'Tate"', 'Sontee', 'revolting.',
'savannah', 'Schlesinger.']
```

Looking at some words from our vocab, what issue do we find by only splitting on spaces?

Your answer here

One large issue we have by splitting only on spaces is that our vocab includes punctuation, single letters, text breakers, names, proper nouns, etc.

Now, rather than naively splitting on spaces, we can use tools which are informed about English grammar rules to create a cleaner tokenization.

In [20]:

```
from nltk.tokenize import word_tokenize

pos_examples_tokenized = [word_tokenize(ex) for ex in pos_examples]
neg_examples_tokenized = [word_tokenize(ex) for ex in neg_examples]

print(pos_examples_tokenized[0])
```

```
['Bromwell', 'High', 'is', 'a', 'cartoon', 'comedy', '.', 'It', 'ran', 'at', 'the', 'sam
e', 'time', 'as', 'some', 'other', 'programs', 'about', 'school', 'life', ',', 'such',
'as', '``', 'Teachers', "''", '.', 'My', '35', 'years', 'in', 'the', 'teaching', 'profes
sion', 'lead', 'me', 'to', 'believe', 'that', 'Bromwell', 'High', "'s", 'satire', 'is',
'much', 'closer', 'to', 'reality', 'than', 'is', '``', 'Teachers', "''", '.', 'The', 'sc
ramble', 'to', 'survive', 'financially', ',', 'the', 'insightful', 'students', 'who', 'c
an', 'see', 'right', 'through', 'their', 'pathetic', 'teachers', "'", 'pomp', ',', 'th
e', 'pettiness', 'of', 'the', 'whole', 'situation', ',', 'all', 'remind', 'me', 'of', 't
he', 'schools', 'I', 'knew', 'and', 'their', 'students', '.', 'When', 'I', 'saw', 'the',
'episode', 'in', 'which', 'a', 'student', 'repeatedly', 'tried', 'to', 'burn', 'down',
'the', 'school', ',', 'I', 'immediately', 'recalled', '.........', 'at', '..........',
'High', '.', 'A', 'classic', 'line', ':', 'INSPECTOR', ':', 'I', "'m", 'here', 'to', 'sa
ck', 'one', 'of', 'your', 'teachers', '.', 'STUDENT', ':', 'Welcome', 'to', 'Bromwell',
'High', '.', 'I', 'expect', 'that', 'many', 'adults', 'of', 'my', 'age', 'think', 'tha
t', 'Bromwell', 'High', 'is', 'far', 'fetched', '.', 'What', 'a', 'pity', 'that', 'it',
'is', "n't", '!']
```

Looking at the first example we can see that things like apostrophes, periods, "n'ts" and ellipses are better handled.

Let's begin defining features for our model. The simplest features are simply if a word exists or not -- however, this is will be very slow if we decide to use the whole vocabulary. Instead, let's create these features for the top 100 most common words.

In [14]:
```
all_tokenized_words = [word for ex in pos_examples_tokenized for word in ex] + \
    [word for ex in neg_examples_tokenized for word in ex]

atw_counter = Counter(all_tokenized_words)
top100 = [tup[0] for tup in atw_counter.most_common(100)] # A list of the top 100 most

print(top100)
```

```
['the', ',', '.', 'and', 'a', 'of', 'to', 'is', '/', '>', '<', 'br', 'in', 'I', 'it', 't
hat', "'s", 'this', 'was', 'The', 'as', 'with', 'movie', 'for', 'film', ')', '(', 'but',
"n't", "''", '``', 'on', 'you', 'are', 'not', 'have', 'his', 'be', 'he', '!', 'one', 'a
t', 'by', 'all', 'an', 'who', 'they', 'from', 'like', 'It', 'her', 'so', 'or', 'about',
'has', 'just', 'out', '?', 'do', 'This', 'some', 'good', 'more', 'very', 'would', 'wha
t', 'there', 'up', 'can', 'which', 'when', 'time', 'she', 'had', 'if', 'only', 'really',
'story', 'were', 'their', 'even', 'see', 'no', 'my', 'me', 'does', "'", 'did', ':', '-',
'than', '...', 'much', 'been', 'could', 'into', 'get', 'will', 'we', 'other']
```

In [ ]:
```

```

In [46]:
```
####################
####################
##   Here I further clean the vocab to improve my model
#############
```

In [ ]:
```

```

In [61]:
```
from nltk.corpus import stopwords

#Function to remove all stop words from the given word list such as "the", "a", "is", e
def RemoveStopWords(word_list):
    filtered_words = [word for word in word_list if word not in stopwords.words('englis
    return filtered_words
```

In [62]:
```
pos_nostop = [RemoveStopWords(ex) for ex in pos_examples_tokenized]
neg_nostop = [RemoveStopWords(ex) for ex in neg_examples_tokenized]
```

In [89]:
```
# Function to remove the proper nouns from a word list
from nltk import pos_tag
def RemoveProperNouns(word_list):
    tagged = nltk.tag.pos_tag(word_list)
    edited = [word for word,tag in tagged if tag != 'NNP' and tag != 'NNPS']
    return edited
```

In [65]:
```
pos_clean = [RemoveProperNouns(ex) for ex in pos_nostop]
neg_clean = [RemoveProperNouns(ex) for ex in neg_nostop]
```

In [ ]:

In [66]:

```
all_clean_words = [word for ex in pos_clean for word in ex] + \
    [word for ex in neg_clean for word in ex]

all_clean_counter = Counter(all_clean_words)
top100clean = [tup[0] for tup in all_clean_counter.most_common(100)] # A list of the to

print(top100clean)
```

```
[',', '.', 'br', 'I', "'s", 'movie', 'The', 'film', ')', '(', "n't", '``', "''", 'one',
'!', '<', 'like', 'It', 'story', '?', 'good', '>', 'time', 'This', 'even', 'would',
'...', 'see', 'really', 'much', 'first', '-', ':', 'people', 'could', 'bad', 'But', 'mov
ies', 'many', 'well', 'think', 'make', 'great', 'scene', 'action', 'get', '&', 'scenes',
"'", 'In', 'characters', 'watch', 'made', 'love', 'never', 'films', 'seen', 'go', 'bette
r', 'little', 'And', 'plot', ';', 'life', 'two', 'way', 'old', 'actors', 'ever', 'charac
ter', 'say', 'director', '2', 'still', 'best', 'got', 'acting', 'every', 'ship', "'ve",
'years', 'know', 'give', 'A', 'man', 'though', 'real', 'nothing', 'He', 'something', 'al
so', 'watching', 'There', 'thought', 'If', 'thing', 'find', 'going', 'woman', 'things']
```

Use the following block to define your own features for the NB model.

In [88]:

```
## I did the cleaning for this added feature above
def top100_clean_word_features(example): # 100 features, 1 for each word in the top 100
    #note cleaned words has no stop words and no nouns
    return {word : 1 if word in example else 0 for word in top100clean}


##list of words that I would consider have a positive sentiment
pos_sentiment=["fantastic", "wonderful", "incredible", "recommended"]
def example_feature(example): #Delete or modify this
    return {word : 1 if word in example else 0 for word in pos_sentiment}

def create_feature_dictionary(example):
    features = {}
    for feat in [ top100_clean_word_features, example_feature]: #Once you've created yo
        features.update(feat(example))
    return features
```

Now that we've defined our features for our model, we can create our final dataset, which will consist of extracted features and the example label.

We'll also create a *validation* split by taking 20% of the training dataset. Remember, we never use the test set to make modeling decisions (in this case, decisions about features). Experiment with multiple models that make use of different combinations of features. Measure their performance on the validation split to figure out which features are the most helpful (use the show_most_informative_features function). When you've found your final model, evaluate its performance on the held out data.

In [82]:

```
from nltk.classify import NaiveBayesClassifier
import random

# Convert training examples to a set of features.
train = [(create_feature_dictionary(ex), 0) for ex in neg_examples] + \
            [(create_feature_dictionary(ex), 1) for ex in pos_examples]
```

```
random.seed(42)
random.shuffle(train)

split_percent = .2

cutoff = int(split_percent * len(train))

validation_set = train[:cutoff]
training_set = train[cutoff:]

model = NaiveBayesClassifier.train(training_set)
```

In [83]:

```
from nltk.classify.util import accuracy

print('Validation accuracy: {}'.format(accuracy(model, validation_set)))
model.show_most_informative_features(10)
```

```
Validation accuracy: 0.7256
Most Informative Features
               wonderful = 1                   1 : 0      =      4.5 : 1.0
               fantastic = 1                   1 : 0      =      4.1 : 1.0
              incredible = 1                   1 : 0      =      3.1 : 1.0
             recommended = 1                   1 : 0      =      3.0 : 1.0
                     bad = 1                   0 : 1      =      2.9 : 1.0
                 nothing = 1                   0 : 1      =      2.1 : 1.0
                   great = 1                   1 : 0      =      2.0 : 1.0
                    love = 1                   1 : 0      =      1.8 : 1.0
                    ship = 1                   1 : 0      =      1.7 : 1.0
                       ? = 1                   0 : 1      =      1.7 : 1.0
```

Describe the sets of features you've considered, and note down their performance below. What is the final set of features you found?

Your answer here

Since our goal is sentiment analysis, I wanted to evaluate the movies using words that actually indicat positive or negative. So I recleanedthe words to create a new feature.

First, I removed all "stop words" such as “a”, “the”, “is”, “are”, etc. since these do not indicate any sentiment Then I removed proper nouns Lastly I took the top100 of that new set of clean words

This brought the performance from ~61% to ~72%

Then,I also devised a list of words that I would classify as "positive sentiment" and that brought the validation accuracy to ~73%. Obviously, this list is just a start and with more research I could certainly improve this

While I technically only added 2 features, the extraction of stop words and nouns took a significant amount of work and research and improved the accuracy considerably more than other ideas I tried (such as length). I hope I am not deducted points simply because I only added 2 features. I always believe in quality over quantity.

Finally, test your model on the test set.

In [85]:

```python
# Load and process test data
pos_test_examples = read_folder(data_dir + 'test/pos/')
neg_test_examples = read_folder(data_dir + 'test/neg/')

test_set = [(create_feature_dictionary(ex), 0) for ex in neg_test_examples] + \
                [(create_feature_dictionary(ex), 1) for ex in pos_test_examples]
```

In [86]:

```python
print('Test set accuracy: {}'.format(accuracy(model, test_set)))

# Note that we're looking at accuracy -- this is not always the most reliable metric an
```

```
Test set accuracy: 0.7192
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

```python
## Self note: Total time spent on homework: 50+ hours. For next time- get debugging hel
#Ask sooner than later!
```