CSC311 Lab 6: Naive Bayes for Classifying Movie Reviews

In this lab, we will build a naive bayes model to classify positive vs negative movie reviews.

By the end of this lab , you will be able to:

1. Use the "bag of word" representation of text.
2. Build naive bayes classifiers to solve classification problems.
3. Analyze MLE vs Bayesian parameter estimation methods.
4. Use a naive bayes classifier to make predictions.

Please work in groups of 1-2 during the lab.

Submission

Submit a PDF document called `lab06.pdf` on Markus. You may also, optionally, submit your lab06.ipynb file. Your PDF file will be graded, but we will ask the TAs to check your ipynb file in case any solutions are cut off. Annotations will be made in the PDF file. Your PDF file should contain the following:

- Part 1. your creation of the `vocab` list and report of the size of the vocabulary (1 point)
- Part 1. your implementation of the `make_bow` function (1 point)
- Part 2. your implementation of the `naive_bayes_mle` function (2 points)
- Part 2. your implementation of the `naive_bayes_map` function (2 points)
- Part 3. your implementation of the `make_prediction` function (3 points)
- Part 3. your training/validation accuracy of a logistic regression model trained on this data (1 point)

You may produce a PDF document by exporting the Colab document, but be careful to check that the required code and output is not cut off. This method is preferred, since we would be able to more easily help point out issues.

Alternatively, you may create a PDF document that contain the parts that are graded. However, the feedback we are able to provide may be more limited.

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline
```

Acknowledgements

Data is a variation of the one from https://www.kaggle.com/datasets/lakshmi25npathi/imdb-dataset-of-50k-movie-reviews, pre-processed so that only 1000 words are in the training/test set.

Part 1 Data

Start by running these two lines of code to download the data on to Google Colab.

```
# Download tutorial data files.
!wget https://www.cs.toronto.edu/~lczhang/311/lab09/trainvalid.csv
!wget https://www.cs.toronto.edu/~lczhang/311/lab09/test.csv
```

```
--2024-03-25 13:40:28--  https://www.cs.toronto.edu/~lczhang/311/lab09/trainvalid.csv
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1464806 (1.4M) [text/csv]
Saving to: ‘trainvalid.csv’

trainvalid.csv      100%[===================>]   1.40M  --.-KB/s    in 0.1s

2024-03-25 13:40:29 (10.2 MB/s) - ‘trainvalid.csv’ saved [1464806/1464806]

--2024-03-25 13:40:29--  https://www.cs.toronto.edu/~lczhang/311/lab09/test.csv
Resolving www.cs.toronto.edu (www.cs.toronto.edu)... 128.100.3.30
Connecting to www.cs.toronto.edu (www.cs.toronto.edu)|128.100.3.30|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 491538 (480K) [text/csv]
Saving to: ‘test.csv’

test.csv            100%[===================>] 480.02K  --.-KB/s    in 0.1s

2024-03-25 13:40:29 (4.22 MB/s) - ‘test.csv’ saved [491538/491538]
```

As always, we start by understanding what our data looks like. Notice that the test set has been set aside for us. Both the training and test set files follow the same format, where each line in the csv file contains the review text and the string label "positive" or "negative".

```
import csv
trainfile = "trainvalid.csv"
testfile = "test.csv"

# Training/Validation set
data = csv.reader(open(trainfile))
```

```
for i, line in enumerate(data):
    print(line)
    if i > 10:
        break

pos, neg = 0, 0
for i, line in enumerate(data):
    if(line[1] =='positive'):
       pos += 1
    else:
      neg += 1

print(pos, neg)
print(neg_count)
```

```
['im not sure i havent seen every episode but i still enjoyed it its hard to say whic
['this film has everything ask yourself are you a fan of and ive seen a lot of movies
['its not going to be the japanese version the show is great but', 'positive']
['i would love to see either of them in another movie', 'positive']
['this movie is great if you enjoy watching b class movies that is', 'positive']
['if you can do that it really is good s s s s i doubt it too bad', 'positive']
['this movie has some things that are pretty amazing first it is supposed to be based
['first of all i want to point on screen play', 'positive']
['this is the best movie i have ever seen any movie that can get children to learn hi
['lets start with the good things this is not a movie for kids the action scenes are
['but it is still dont care is it worth try it', 'positive']
['while by no means a classic the directors involved do have an idea what suspense is
5988 6000
0
```

```
# Test set (separated so that we will all have the same test set)
data = csv.reader(open(testfile))
pos, neg = 0, 0
for i, line in enumerate(data):
    print(line)
    if i > 10:
        break

for i, line in enumerate(data):
    if(line[1] =='positive'):
       pos += 1
    else:
      neg += 1

print(pos, neg)
```

```
['what a great film a great cast as well i really enjoy this movie', 'positive']
['i cant say enough about this movie sometimes it feels as if youre watching a real d
['it was probably the most black comedy ive seen i really do recommend it to you', 'p
['old movies new movies im not sure why i cant get a copy its about time for this to
['that film is absolutely fantastic if you watch it with your friends it can be a ver
['one did not need it anyway she knows this man is different', 'positive']
['he was something never before seen he was all we ever needed', 'positive']
```

```
['lets take the second point first but it is only that moment all three white actors
['i also think this is one of the greatest movies of the last years', 'positive']
['a great movie the acting is basically above average nothing special but better then
['if you get a chance watch this movie and it is family comedy entertainment at its b
['i remember watching this movie when it came out as a t they just dont make t all on
1988 2000
```

Task: How many positive reviews are in `trainvalid.csv`? Negative reviews? What about in `test.csv`?

```
# trainvalid.csv has 5988 positive, 6000 negative.
# test.csv has 1988 positive, 2000 negative.
```

Graded Task: How many unique words are in the entire training/validation set? This will be the size of our **vocabulary** when we convert to a bag-of-word representation. To compute this value, produce a list `vocab` that contains a list of unique words in `trainvalid.csv`.

Splitting sentences into words is typically not a trivial operation. For this lab, punctuations and other special symbols have already been removed from the reviews. So, using python's `string.split()` method is sufficient.

```
vocab = set()

data = csv.reader(open(trainfile))
for i, line in enumerate(data):
    vocab.update(line[0].split(" "))

print("Vocabulary Size: ", len(vocab)) # Please include the output of this statement in y
```

```
 Vocabulary Size:  1000
```

Graded Task: Complete the function `make_bow`, which takes a list of `(review, label)` pairs and a list of words in the vocabulary, and produces a data matrix consisting of bag-of-word features, along with a vector of labels.

```
def make_bow(data, vocab):
    """
    Produce the bag-of-word representation of the data, along with a vector
    of labels. You *may* use loops to iterate over `data`. However, your code
    should not take more than O(len(data) * len(vocab)) to run.

    Parameters:
        `data`: a list of `(review, label)` pairs, like those produced from
                `list(csv.reader(open("trainvalid.csv")))`
        `vocab`: a list consisting of all unique words in the vocabulary
```

```
    Returns:
        `X`: A data matrix of bag-of-word features. This data matrix should be
             a numpy array with shape [len(data), len(vocab)].
             Moreover, `X[i,j] == 1` if the review in `data[i]` contains the
             word `vocab[j]`, and `X[i,j] == 0` otherwise.
        `t`: A numpy array of shape [len(data)], with `t[i] == 1` if
             `data[i]` is a positive review, and `t[i] == 0` otherwise.
    """
    X = np.zeros([len(data), len(vocab)])
    t = np.zeros([len(data)])

    # TODO: fill in the appropriate values of X and t

    for i, (review, label) in enumerate(data):
        # Set the label
        t[i] = int(label == 'positive')

        # Split the review into words
        review_words = set(review.split(" "))

        # An array representing whether j-th word in vocab is in the datapoint
        X[i, :] = np.array([word in review_words for word in vocab])

    return X, t

# Separate data into training/validation, then call `make_bow` to produce
# the bag-of-word features
import random
random.seed(42)
data = list(csv.reader(open(trainfile)))
random.shuffle(data)
X_train, t_train = make_bow(data[:10000], vocab)
X_valid, t_valid = make_bow(data[10000:], vocab)
```

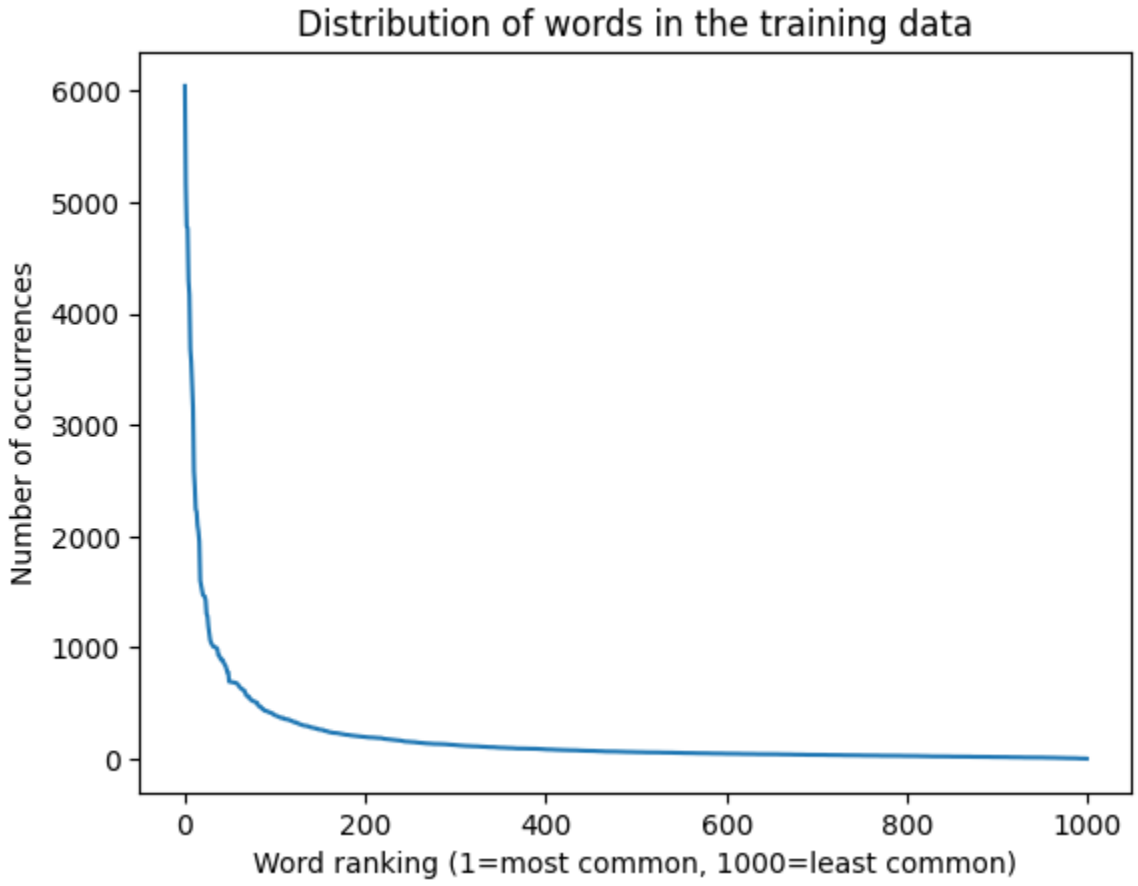
Task: It is a good idea to understand the distribution of relative word occurrences. Produce the figure below, which shows how often each word occurs in the training set.

```
# produce the mapping of words to count
vocab_count_mapping = list(zip(vocab, np.sum(X_train, axis=0)))
vocab_count_mapping = sorted(vocab_count_mapping, key=lambda e: e[1], reverse=True)
for word, cnt in vocab_count_mapping:
    print(word, cnt)
```

If we plot these occurrences, we see that the more common words tend to occur much more frequently compared to the less common words. This type of distribution occurs quite often in typical data sets, where there are a **long tail** of words that occur infrequently.

```
plt.plot([cnt for word, cnt in vocab_count_mapping])
plt.title("Distribution of words in the training data")
plt.xlabel("Word ranking (1=most common, 1000=least common)")
plt.ylabel("Number of occurrences")
```

```
Text(0, 0.5, 'Number of occurrences')
```



Part 2. Naive Bayes Model

In this section, we will build a naive bayes model to predict whether a movie review is positive or negative. The naive bayes model is a generative model and a probabilistic classifier, where we will make the **conditional independence assumption**:

$$p(c, x_1, \ldots, x_D) = p(c)p(x_1|c) \cdots p(x_D|c)$$

Where c is the label class (positive or negative), and each $x_j \in 0, 1$ represents whether word j appears in the review. We will fit parameters π and θ_{jc}, with:

- $p(c = 1) = \pi$, representing the probability of positive class
- $p(x_j = 1|c) = \theta_{jc}$, representing the probability of word j appearing in a review with class c.

Graded Task: Implement the function `naive_bayes_mle` that computes the MLE estimation of parameters. This function will produce the estimates of π and the θ_{jc}s that optimizes the log

likelihood of the training data:

$$\ell(\theta) = \sum_{i=1}^{N} \log p(c^{(i)}, \mathbf{x}^{(i)})$$

Like mentioned in class, this formula sounds scary, but the estimates of π and θ_{jc} turns out to require only basic counting operations and division. As such, we can implement this function in an entirely vectorized way, **without using loops**. Please do not use any loops in your solution.

```
def naive_bayes_mle(X, t):
    """
    Compute the parameters $pi$ and $theta_{jc}$ that maximizes the log-likelihood
    of the provided data (X, t).

    **Your solution should be vectorized, and contain no loops**

    Parameters:
        `X` - a matrix of bag-of-word features of shape [N, V],
              where N is the number of data points and V is the vocabulary size.
              X[i,j] should be either 0 or 1. Produced by the make_bow() function.
        `t` - a vector of class labels of shape [N], with t[i] being either 0 or 1.
              Produced by the make_bow() function.

    Returns:
        `pi` - a scalar; the MLE estimate of the parameter $\pi = p(c = 1)$
        `theta` - a matrix of shape [V, 2], where `theta[j, c]` corresponds to
                  the MLE estimate of the parameter $\theta_{jc} = p(x_j = 1 | c)$
    """
    N, vocab_size = X.shape[0], X.shape[1]
    pi = np.mean(t) # Probability that p(c = 1). Each elem. in t is in {0, 1}. Take
    theta = np.zeros([vocab_size, 2])

    # these matrices may be useful (but what do they represent?)
    # The occurrences of each positive/negative word in just that datapoint
    X_positive = X[t == 1]
    X_negative = X[t == 0]

    # #occurrences of type in data / #total of type in data
    theta[:, 1] = np.sum(X_positive) / np.sum(t == 1)
    theta[:, 0] = np.sum(X_negative) / np.sum(t == 0)

    return pi, theta

pi_mle, theta_mle = naive_bayes_mle(X_train, t_train)

print(theta_mle.shape) # should be (1000, 2)
```

```
(1000, 2)
```

Graded Task: Implement the function `naive_bayes_map` that computes the MAP estimation of parameters. This function will produce the estimates of π and the θ_{jc}s that maximizes the posterior: $p(\theta|\mathcal{D})$, where $\theta = \pi, \theta_{jc}$ consists of all of our parameters.

We will use the beta distribution with $a = 2$ and $b = 2$ for all of our parameters.

Once again, although these words might sound scary, the estimates of π and θ_{jc} turns out to require only basic counting operations and division---and not much more than the previous part! Again, we can implement this function in an entirely vectorized way, **without using loops**. Please do not use any loops in your solution.

```
def naive_bayes_map(X, t):
    """
    Compute the parameters $pi$ and $theta_{jc}$ that maximizes the posterior
    of the provided data (X, t). We will use the beta distribution with
    $a=2$ and $b=2$ for all of our parameters.

    **Your solution should be vectorized, and contain no loops**

    Parameters:
        `X` - a matrix of bag-of-word features of shape [N, V],
              where N is the number of data points and V is the vocabulary size.
              X[i,j] should be either 0 or 1. Produced by the make_bow() function.
        `t` - a vector of class labels of shape [N], with t[i] being either 0 or 1.
              Produced by the make_bow() function.

    Returns:
        `pi` - a scalar; the MAP estimate of the parameter $\pi = p(c = 1)$
        `theta` - a matrix of shape [V, 2], where `theta[j, c]` corresponds to
                  the MAP estimate of the parameter $\theta_{jc} = p(x_j = 1 | c)$
    """
    N, V = X.shape
    a, b = 2, 2

    pi = (np.sum(t) + a - 1) / (N + a + b - 2)
    theta = np.zeros((V, 2))

    theta_pos = (np.dot(X.T, t) + a - 1)
    theta_neg = (np.dot(X.T, 1 - t) + a - 1)
    denom = (np.sum(t) + a + b - 2)
    theta[:, 1] = theta_pos / denom
    theta[:, 0] = theta_neg / denom

    return pi, theta


pi_map, theta_map = naive_bayes_map(X_train, t_train)
```

```
print(theta_map.shape) # should be (1000, 2)
```

```
(1000, 2)
```

Part 3. Making predictions

Graded Task: Complete the function `make_prediction` which uses our estimated parameters π and θ_{jc} to make predictions on our dataset.

Note that computing products of many small numbers leads to underflow. Use the fact that:

$$a_1 \cdot a_2 \cdots a_n = e^{log(a_1)+log(a_2)+\cdots+log(a_n)}$$

to avoid computing a product of small numbers.

```
def make_prediction(X, pi, theta):

    log_theta_1 = np.log(theta[:, 1])
    log_1_minus_theta_1 = np.log(1 - theta[:, 1])

    log_theta_0 = np.log(theta[:, 0])
    log_1_minus_theta_0 = np.log(1 - theta[:, 0])

    log_likelihood_0 = np.dot(X, log_1_minus_theta_1) + np.dot(1 - X, log_theta_1)
    log_likelihood_1 = np.dot(X, log_1_minus_theta_0) + np.dot(1 - X, log_theta_0)

    return log_likelihood_1 > log_likelihood_0

def accuracy(y, t):
    return np.mean(y == t)


y_map_train = make_prediction(X_train, pi_map, theta_map)
y_map_valid = make_prediction(X_valid, pi_map, theta_map)

print("MAP Train Acc:", accuracy(y_map_train, t_train))
print("MAP Valid Acc:", accuracy(y_map_valid, t_valid))
```

```
MAP Train Acc: 0.4993
MAP Valid Acc: 0.5035
```

At this point, you might wonder if the accuracy is "about right", and you might ask your instructors/TAs about what the expected accuracy might be. But what can we do to verify our results in real applications? One strategy is to compare your model output with other known models.

Graded Task: Use sklearn to build a logistic regression model. Report the training and validation

accuracy. You should see that the Logistic Regression model performs a bit better than the Naive Bayes model. The Logistic Regression model does not make as many assumptions about the data distribution that the Naive Bayes does. (Think: what are these assumptions?)

```
from sklearn.linear_model import LogisticRegression

lr_model = LogisticRegression(max_iter=500)

lr_model.fit(X_train, t_train)

y_train_pred = lr_model.predict(X_train)
y_valid_pred = lr_model.predict(X_valid)

train_acc = accuracy(y_train_pred, t_train)
val_acc = accuracy(y_valid_pred, t_valid)

print("LR Train Acc:", train_acc)
print("LR Valid Acc:", val_acc)
```

```
LR Train Acc: 0.8249
LR Valid Acc: 0.7865
```

This task is also a reminder that the bag of word features we built are just *features*, and can be used with *any* model!

Task: The below code produces a warning or an error. Why?

```
y_mle_train = make_prediction(X_train, pi_mle, theta_mle)
y_mle_valid = make_prediction(X_valid, pi_mle, theta_mle)
print("MLE Train Acc:", accuracy(y_mle_train, t_train))
print("MLE Valid Acc:", accuracy(y_mle_valid, t_valid))
```

Task: The function `predict` is written for you, and provides another way to interact with the Naive Bayes model. The function takes a review (as a string), and relevant information about the naive bayes model, and produce an estimate for that single string.

Call `predict` several times, with short reviews that you write, to get a sense of how the model behaves.

```
def predict(review, vocab, pi, theta):
    x = np.zeros([1, len(vocab)])
    words = review.split()
    for j, w in enumerate(vocab):
        if w in words:
            x[0, j] = 1
    return make_prediction(x, pi, theta)[0]
```

```
predict("movie was fun", vocab, pi_map, theta_map)
```

Task: Explain why the MLE model fails to make predictions (or produces a warning) for a review with the word "william" in it.

```
print(predict("william", vocab, pi_map, theta_map)) # why does this succeed
print(predict("william", vocab, pi_mle, theta_mle)) # ...and this produce an error?

# Your explanation goes here
```

Task: Report the test accuracy on the naive bayes model with parameter estimation via MAP.

```
# TODO
```

Part 4. Analyzing the MAP Parameters

Task: Write code to print the following:

- the 10 words whose *presence* most strongly predicts that the review is positive
- the 10 words whose *absense* most strongly predicts that the review is positive
- the 10 words whose *presence* most strongly predicts that the review is negative
- the 10 words whose *absense* most strongly predicts that the review is negative

Note that these require more than just sorting the θ_{jc}s! (Why?)