CS114 (Spring 2020) Programming Assignment 2
Naïve Bayes Classifier and Evaluation

Due February 25, 2020

You are given `naive_bayes.py`, and `movie_reviews.zip`, the NLTK movie review corpus. Reviews are separated into a training set (80% of the data) and a development set (10% of the data). A testing set (10% of the data) has been held out and is not given to you. Within each set, reviews are sorted by sentiment (positive/negative). The files are already tokenized. Each review is in its own file. You are also given `movie_reviews_small.zip`, the toy corpus from HW1, in the same format.

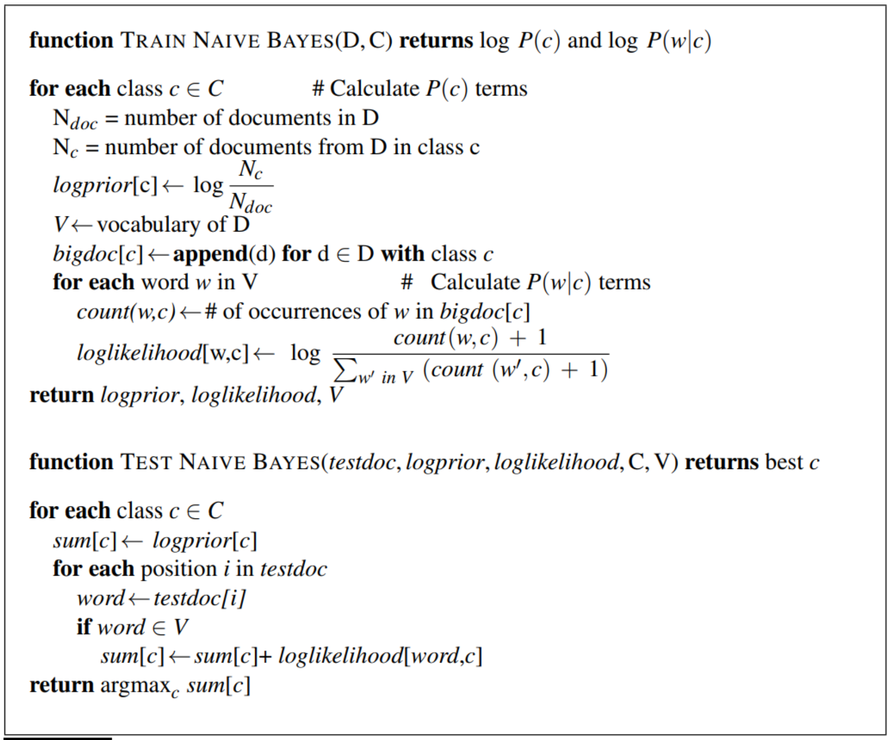
You will need to use Numpy for this assignment. A description of useful Numpy functions is in the Appendix.

Assignment

Your task is to implement a multinomial Naïve Bayes classifier using bag-of-words features. Specifically, in `naive_bayes.py`, you should fill in the following functions:

- `train(self, train_set)`: This function should, given a folder of training documents, fill in `self.prior` and `self.likelihood`, such that:
  - `self.prior[class]` $= \log(P(\text{class}))$
  - `self.likelihood[class][feature]` $= \log(P(\text{feature}|\text{class}))$

  You can use the pseudo-code given in Figure 4.2 of the Jurafsky and Martin book, reproduced below.



```
function TRAIN NAIVE BAYES(D, C) returns log $P(c)$ and log $P(w|c)$

for each class $c \in C$              # Calculate $P(c)$ terms
    $N_{doc}$ = number of documents in D
    $N_c$ = number of documents from D in class c
    $logprior[c] \leftarrow \log \frac{N_c}{N_{doc}}$
    $V \leftarrow$ vocabulary of D
    $bigdoc[c] \leftarrow$ append(d) for d $\in$ D with class $c$
    for each word $w$ in V                # Calculate $P(w|c)$ terms
        $count(w,c) \leftarrow$ # of occurrences of $w$ in $bigdoc[c]$
        $loglikelihood[w,c] \leftarrow \log \frac{count(w,c) + 1}{\sum_{w' \text{ in } V} (count(w',c) + 1)}$
return $logprior$, $loglikelihood$, $V$

function TEST NAIVE BAYES($testdoc$, $logprior$, $loglikelihood$, C, V) returns best $c$

for each class $c \in C$
    $sum[c] \leftarrow logprior[c]$
    for each position $i$ in $testdoc$
        $word \leftarrow testdoc[i]$
        if $word \in V$
            $sum[c] \leftarrow sum[c] + loglikelihood[word,c]$
return $\text{argmax}_c\ sum[c]$
```

Figure 4.2 The naive Bayes algorithm, using add-1 smoothing. To use add-α smoothing instead, change the $+1$ to $+\alpha$ for loglikelihood counts in training.

`self.prior` and `self.likelihood` should be Numpy arrays, `self.prior` a vector (array of rank 1) of shape $(|C|,)$, and `self.likelihood` a matrix (array of rank 2) of shape $(|C|, |F|)$, where $|C|$ and $|F|$ are the numbers of classes and features, respectively. `self.class_dict` and `self.feature_dict` should be used to translate between indices and class/feature names. For example, if `self.class_dict[0]` is negative, and `self.feature_dict[1]` is "couple", then `self.likelihood[0][1]` should be $\log(P(\text{couple}|\text{negative}))$. Importantly, note that although we are not using the entire vocabulary as features, the V in the denominator of the log-likelihood term should still be the entire vocabulary; i.e., the denominator is the total count of all words, feature or not, in documents of class c.

- `test(self, dev_set)`: This function should, given a folder of development (or testing) documents, return a dictionary of `results` such that:

    - `results[filename]['correct']` = correct class
    - `results[filename]['predicted']` = predicted class

  You can look at the pseudo-code in Figure 4.2 of the book for inspiration, but our procedure will be somewhat different. For each document, we will create a feature vector: if `self.feature_dict[1]` is "couple", then the second element of the vector will be the count of how many times "couple" appears in the document. We can then take the matrix product of `self.likelihood` and our vector: the product of our $|C| \times |F|$ log-likelihood matrix and our feature vector of length $|F|$ will be a vector of length $|C|$ that contains, for each class, the log-likelihood of the document given the class. We can then add `self.prior` to this vector and take the argmax to find the most probable class.

- `evaluate(self, results)`: This function should, given the results of `test`, compute precision, recall, and F1 score for each class, as well as the overall accuracy, and print them in a readable format. Recall that (where c_{ij} is the number of documents actually in class i that were classified as being in class j):

    - precision $= \dfrac{c_{ii}}{\sum\limits_j c_{ji}}$
    - recall $= \dfrac{c_{ii}}{\sum\limits_j c_{ij}}$
    - F1 $= \dfrac{2 \times \text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$
    - accuracy $= \dfrac{\sum\limits_i c_{ii}}{\sum\limits_j \sum\limits_i c_{ij}}$

  When calculating your evaluation metrics, it may be helpful to use a confusion matrix. The `confusion_matrix` defined in `evaluate` can be populated as follows: `confusion_matrix[class_1][class_2]` = the number of documents classified as `class_1` that are actually in `class_2`.

- `select_features(self, train_set)`: Congratulations, you have implemented your very own Naive Bayes classifier! At this point, you should experiment with additional features. Perhaps the word "shoot", while a good feature to distinguish comedies and action movies, might not be as good for distinguishing positive and negative movie reviews.

  You can use almost any method you want to select features. For example, one possibility is to compute the mutual information for each word:

$$\begin{aligned} I(w) = &-\sum_{c \in C} P(c) \log P(c) \\ &+ P(w) \sum_{c \in C} P(c|w) \log P(c|w) \\ &+ P(\bar{w}) \sum_{c \in C} P(c|\bar{w}) \log P(c|\bar{w}) \end{aligned}$$

  (Note that this method treats the presence of a word as a Bernoulli random variable, so that $P(w)$ is the probability that w appears in a document, $P(\bar{w})$ is the probability that w does not appear in a document, and likewise for the conditional probabilities.)

  Another possibility is to compute the likelihood ratio (NLTK does this) for each word as follows:

$$LR(w) = \max_{i,j} \frac{P(w|c_i)}{P(w|c_j)}$$

  The simplest possibility, of course, is trial and error. The only thing you cannot do, is to use an outside sentiment lexicon; we want you to do your own feature selection. Turn in your model that performs best on the development set.

Write-up

You should also prepare a short write-up that includes at least the following:

- What additional features you included/tried in your classifier
- Your evaluation results on the development set (you do not need to include any results on the toy data)

Submission Instructions

Please submit two files: your write-up (in PDF format), and `naive_bayes.py`. Do not include any data.

Appendix: Useful Numpy functions

You may find the following Numpy functions useful:

- `numpy.zeros(shape)`: Returns an array of given `shape`, filled with zeros.
- `numpy.log(x)`: If `x` is a number, returns the (natural) log of `x`. If `x` is an array, returns an array containing the logs of each element of `x`.
- `numpy.dot(a, b)`: Returns the product of `a` and `b`, where the type of product depends on the shapes of `a` and `b`.
- `numpy.argmax(a, axis=None)`: Given an array `a`, returns the argmax(es) along an axis. If no axis is given, returns the argmax over the entire array (flattening it into a vector if necessary).
- `numpy.sum(a, axis=None)`: Given an array `a`, returns the sum(s) over an axis. If no axis is given, returns the sum over the entire array.
- `numpy.trace(a)`: Given a matrix `a`, returns the sum along the diagonal.