MSDS 621 - Introduction to Machine Learning

Homework 2

Robert Clements

1 Sentiment Analysis of Movie Reviews with Naive Bayes

1.1 Turning it in

We will use Github classroom to submit assignments. For this assignment you must submit one file: - bayes.py file

1.2 Goal

In this project, you will build a multinomial naive bayes classifier to predict whether a movie review is positive or negative. As part of the project, you will also learn to do k-fold cross validation testing.

1.3 Getting started

Download and uncompress polarity data set v2.0 into the root directory of your repository, but **do not add the data to your git repository**. My directory looks like:

```
$ ls
bayes.py   review_polarity/  test_bayes.py
```

`review_polarity/` is a folder and has the data in it that you need.

The starter code in the repository will help get you started with the assignment.

1.4 Discussion

1.4.1 Naive bayes classifiers for text documents

A text classifier predicts to which class, c, an unknown document d belongs. In our case, the predictions are binary: $c = 0$ for negative movie review and $c = 1$ for positive movie review. We can think about classification mathematically as picking the most likely class:

$$c^* = \underset{c}{argmax}\ P(c|d)$$

We can replace $P(c|d)$, using Bayes' theorem:

$$P(c|d) = \frac{P(c)P(d|c)}{P(d)}$$

to get the formula

$$c^* = \underset{c}{argmax} \frac{P(c)P(d|c)}{P(d)}$$

Since $P(d)$ is a constant for any given document d, we can use the following equivalent but simpler formula to find the most likely (even though the scalar $P(c)P(d|c)$ will not technically be a probability):

$$c^* = \underset{c}{argmax}\ P(c)P(d|c)$$

Training then consists of estimating $P(c)$ and $P(c|d)$, which we'll get to shortly.

1.4.2 Representing documents

Text classification requires a representation for document d. When loading a document, we first load the text and then *tokenize* the words, stripping away punctuation and stop words like *the*. The list of words is a fine representation for a document except that each document has a different length, which makes training most models problematic as they assume tabular data with a fixed number of features. The simplest and most common approach is to create an overall vocabulary, V, created as a set of unique words across all documents in all classes. *Sort the unique words in the vocabulary alphabetically so we standardize which word is associated with which word vector index.* Then, the training features (columns of your X matrix) are those words.

One way to represent a document then is with a binary word vector, with a 1 in each column if that word is present in the document. Then we could perform Naive Bayes using the documents represented in this way (this is the Bernoulli version of Naive Bayes). Something like this:

```
import pandas as pd
df = pd.DataFrame(data=[[1,1,0,0],
                        [0,0,1,1]],
                  columns=['cat','food','hong','kong'])
df
```

	cat	food	hong	kong
0	1	1	0	0
1	0	0	1	1

This tends to work well for very short strings/documents, such as article titles or tweets. For longer documents, using a binary feature loses a lot of information. Instead, it's better to count the number of times each word is present (this is the Multinomial version of Naive Bayes). For example, here are 3 documents and resulting word vectors:

```
d1 = "cats food cats cats"
d2 = "hong kong hong kong"
d3 = "cats in hong kong"  # assume we strip stop words like "in"
df = pd.DataFrame(data=[[3,1,0,0],
                        [0,0,2,2],
                        [1,0,1,1]],
                  columns=['cat','food','hong','kong'])
df
```

	cat	food	hong	kong
0	3	1	0	0
1	0	0	2	2
2	1	0	1	1

These word vectors with fixed lengths are how most models expect data, including sklearn's implementation. Here's how to train a Naive Bayes model with sklearn using the trivial/toy `df` data and get the training set error:

```
from sklearn.naive_bayes import MultinomialNB
import numpy as np
```

```
X = df.values
y = [0, 1, 1] # assume document classes
sknb = MultinomialNB()
sknb.fit(X, y)
y_pred = sknb.predict(X)
print(f"Correct = {np.sum(y==y_pred)} / {len(y)} = {100*np.sum(y==y_pred) / len(y):.1f}%")
```

```
Correct = 3 / 3 = 100.0%
```

Because it is convenient to keep word vectors in a 2D matrix and it is what sklearn likes, we will use the same representation in this project. Given the directory name, your function `load_docs()` will return a list of word lists where each word list is the raw list of tokens, typically with repeated words. Then, function `vocab()` will create the combined vocabulary as a mapping from word to word-feature-index, starting with index 1. Index 0 is reserved for unknown words. Vocabulary V should be a `defaultintdict()`, which I provided, so that unknown words get mapped to value/index 0. Finally, function `vectorize()` will convert that to a 2D matrix, one row per document (called within the function `vectorize_docs()` below):

```
neg = load_docs(neg_dir)
pos = load_docs(pos_dir)
V = vocab(neg,pos)
vneg = vectorize_docs(neg, V)
vpos = vectorize_docs(pos, V)
```

The `defaultintdict` class behaves exactly like `defaultdict(int)` except `d['foo']` does **not** add 'foo' to dictionary `d`.

```
class defaultintdict(dict):
    def __init__(self): # Create dictionary of ints
        self._factory=int
        super().__init__()

    def __missing__(self, key):
        "Override default behavior so missing returns 0"
        return 0
```

1.4.3 Estimating probabilities

To train our model, we need to estimate $P(c)$ and $P(d|c)$ for all classes and documents. Estimating $P(c)$ is easy: it's just the number of documents in class c divided by the total number of documents. To estimate $P(d|c)$, Naive Bayes assumes that each word is *conditionally independent*, given the class, meaning:

$$P(d|c) = \prod_{w \in d} P(w|c)$$

so that gives us:

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in d} P(w|c)$$

where w iterates through all (non-unique) words in d, so the product includes $P(w|c)$ 5 times if w appears 5 times in d.

Because we are going to use word counts, not binary word vectors, in fixed-length vectors, we need to include $P(w|c)$ explicitly multiple times for repeated w in d:

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in V} P(w|c)^{n_w(d)}$$

where $n_w(d)$ is the number of times w occurs in d, V is the overall vocabulary (set of unique words from all documents); $n_w(d) = 0$ when w isn't present in d.

Now we have to figure out how to estimate $P(w|c)$, the probability of seeing word w given that we're looking at a document from class c. That's just the number of times w appears in all documents from class c divided by the total number of words (including repeats) in all documents from class c:

$$P(w|c) = \frac{wordcount(w, c)}{wordcount(c)}$$

1.4.4 Making predictions

Once we have the appropriate parameter estimates, we have a model that can make predictions in an ideal setting:

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in V} P(w|c)^{n_w(d)}$$

1.4.4.1 Avoiding $P(w|c) = 0$

If word w does not exist in class c (but is in V), then the overall product goes to 0 (and when we take the log below, the classifier would try to evaluate $log(0)$, which is undefined). To solve the problem, we use *Laplace Smoothing*, which just means adding 1 to each word count when computing $P(w|c)$ and making sure to compensate by adding $|V|$ to the denominator (adding 1 for each vocabulary word):

$$P(w|c) = \frac{wordcount(w, c) + 1}{wordcount(c) + |V|}$$

where $|V|$ is the size of the vocabulary for all documents in all classes. Adding this to the denominator, keeps $P(w|c)$ a probability. This way, even if $wordcount(w, c)$ is 0, this ratio $>$ 0. If w is not in any doc of class c then $P(w|c) = 1/(wordcount(c) + |V|)$, which is a very low probability. (Note: Each doc's word vector has length $|V|$. During training, any w not found in docs of c, will have word count 0. When we add $+1$, then c looks like it has every word in V. Hence, we must divide by $|V|$ not $|V_c|$.)

1.4.4.2 Dealing with missing words

Laplace smoothing deals with w that are in the vocabulary V but that are not in a class, hence, giving $wordcount(w, c) = 0$ for some c. There's one last slightly different problem. If a future unknown document contains a word not in V (i.e., not in the training data), then what should $wordcount(w, c)$ be? Probably not 0 because that would mean we had data indicating it does not appear in class c when we have *no* training data on it.

To be strictly correct and keep $P(w|c)$ a probability in the presence of unknown words, all we have to do is add 1 to the denominator in addition to the Laplace smoothing changes:

$$P(w|c) = \frac{wordcount(w, c) + 1}{wordcount(c) + |V| + 1}$$

We are lumping all unknown words into a single "wildcard" word that exists in every V. That has the effect of increasing the overall vocabulary size for class c to include room for an unknown word (and all unknown words map to that same spot). In this way, an unknown word gets probability:

$$P(unknown|c) = \frac{0 + 1}{wordcount(c) + |V| + 1}$$

In the end, this is no big deal as all classes will get the same nudge for the unknown word so classification won't be affected.

To deal with unknown words in the project, we can reserve word index 0 to mean unknown word. All words in the training vocabulary start at index 1. So, if we normally have $|V|$ words in the training vocabulary, we will now have $|V| + 1$; no word will ever have 0 word count. Each word vector will be of length $|V| + 1$.

The `vocab()` function in your project returns $|V| = |uniquewords| + 1$ to handle the unknown word wildcard. Once computed, the size of the vocabulary should never change; all word vectors are size $|V|$.

With this new adjusted estimate of $P(w|c)$, we can simplify the overall prediction problem to use $w \in V$:

$$c^* = \underset{c}{argmax}\ P(c) \prod_{w \in V} P(w|c)^{n_w(d)}$$

That means we can use dot products for prediction, which is faster than iterating in python through unique document words.

1.4.4.3 Floating point underflow

The first problem involves the limitations of floating-point arithmetic in computers. Because the probabilities are less than one and there could be tens of thousands multiplied together, we risk floating-point underflow. That just means that eventually the product will attenuate to zero and our classifier is useless. The solution is simply to take the log of the right hand side because log is monotonic and won't affect the $argmax$:

$$c^* = \underset{c}{argmax} \left\{ log(P(c)) + \sum_{w \in V} log(P(w|c)^{n_w(d)}) \right\}$$

Or,

$$c^* = \underset{c}{argmax} \left\{ log(P(c)) + \sum_{w \in V} n_w(d) \times log(P(w|c)) \right\}$$

1.4.4.4 Speed issues

For large data sets, Python loops often are too slow and so we have to rely on vector operations, which are implemented in C. For example, the `predict(X)` method receives a 2D matrix of word vectors and must make a prediction for each one. The temptation is to write the very readable:

```
y_pred = []
for each row d in X:
    y_pred = prediction for d
return y_pred
```

But, you should use the built-in `numpy` functions such as `np.dot` (same as the `@` operator) and apply functions across vectors. For example, if I have a vector, v, and I'd like the log of each value, don't write a loop. Use `np.log(v)`, which will give you a vector with the results.

My `predict()` method consists primarily of a matrix-vector multiplication per class followed by `argmax`.

1.5 Deliverables

You must implement naive bayes from scratch; you cannot use sklearn's built-in `MultinomialNB` class for your implementation. Of course, we do use that for testing purposes.

To submit your project, ensure that your `bayes.py` file is submitted to your repository. That file must be in the root of your repository. It should not have a main program; it should consist of a collection of functions. You must implement the following functions:

- `load_docs(docs_dirname)`
- `vocab(neg, pos)`
- `vectorize(V, docwords)`
- `vectorize_docs(docs, V)`
- `kfold_CV(model, X, y, k=4)` (You must implement manually; don't use sklearn's version but you **can** use `KFold` as part of your function)

and implement class `NaiveBayes621` with these methods

- `fit(self, X, y)`
- `predict(self, X)`

Remember not to add the data to your git repository.
Please do not use sklearn's vectorizer / counter objects; you must learn to implement word vectorization yourself.

1.6 Evaluation

To evaluate your projects I will download your repository and run `test_bayes.py` from your root directory. Here is a sample test run:

```
$ python -m pytest -v test_bayes.py
platform darwin -- Python 3.12.0, pytest-7.4.3, pluggy-1.3.0 -- /opt/anaconda3/envs/ml/bin/py
cachedir: .pytest_cache
rootdir: /Users/rclements/Documents/teaching/msds621/2022/my_solutions/bayes
plugins: anyio-4.0.0
collected 10 items

test_bayes.py::test_load PASSED                                          [ 10%]
test_bayes.py::test_vocab PASSED                                         [ 20%]
test_bayes.py::test_vectorize_docs PASSED                                [ 30%]
test_bayes.py::test_vectorize PASSED                                     [ 40%]
test_bayes.py::test_simple_docs_error PASSED                             [ 50%]
test_bayes.py::test_unknown_words_vectorize PASSED                       [ 60%]
test_bayes.py::test_unknown_words_training_error PASSED                  [ 70%]
test_bayes.py::test_training_error PASSED                                [ 80%]
test_bayes.py::test_kfold_621 PASSED                                     [ 90%]
test_bayes.py::test_kfold_sklearn_vs_621 PASSED                          [100%]

============================== 10 passed in 16.75s ==============================
```

Notice that it takes about 17 seconds. If your project takes more than 45 seconds, I will take off **10 points** from 100. Do not use the timing from Github actions, it usually takes a little longer to run there. The grader will pull your repositories and run the unit tests on their machine in order to evaluate them.

For full credit, you must get all 10 unit tests passing *most of the time* (at least 90%). If any test fails more often, then there is most likely still something wrong with your code. I suggest that you run your tests both locally **and** with github actions (described in the next section) so that you can verify your code works on a different computer. Late projects get a zero grade. The failure of the unit tests will be used as a guide to determine your points. In general, each failed test may result in minus **7 points**.

I have created a hidden test using a totally different data set for this project that has **3 extra tests, each of which is worth 5%**. In other words the best you can do without passing the unknown tests is 85%.

1.6.1 Automatic testing using github actions

With Github Actions every push to the repository on github from your laptop triggers the unit tests. All you have to do is put the test.yml file I have prepared for you into repo subdirectory .github/workflows, commit, and push back to github. Then go to the Actions tab of your repository.

Naturally it will only work if you have your software written and added to the repository. Once you have something basic working, this functionality is very nice because it automatically shows you how your software is going to run on a different computer (a linux computer). This will catch the usual errors where you have hardcoded something from your machine into the software. It also gets you in the habit of committing software to the repository as you develop it, rather than using the repository as a homework submission device.