Implementing Naive Bayes Classifiers

Terence Parr

July 13, 2019

1 Movie reviews sentiment analysis with Naive Bayes

Project 2, MS621 *Fall 2019*

1.1 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.

You will do your work in a `bayes-`*userid* repository. Please keep all of your files in the root directory of the repository.

1.2 Getting started

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

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

Download the test_bayes.py script into the root directory of your repository; you can add this if you want but I will overwrite it when testing. It assumes that the `review_polarity` directory is in the same directory (the root of the repository).

Download the bayes.py starter kit into the root directory of your repository. Make sure to add this to the repo.

See Naive Bayes discussion, p258 in Introduction to Information Retrieval.

Please do not add the data to your repository!

1.3 Discussion

1.3.1 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 getg 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:

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

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

1.3.2 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. Then, the training features are those words.

One way to represent a document than is with a binary word vector, with a 1 in each column if that word is present in the document. Something like this:

```
In [3]: import pandas as pd
        df = pd.DataFrame(data=[[1,1,0,0],
                                [0,0,1,1]], columns=['cat','food','hong','kong'])
        df
```

```
Out[3]:    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 presence or absence loses information. Instead, it's better to count the number of times each word is present. For example, here are 3 documents and resulting word vectors:

```
In [6]: 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
```

```
Out[6]:    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. (It's assuming Gaussian distributions for probability estimates where as we are assuming multinomial, but we can still shove our data in.) Here's how to train a Naive Bayes model with sklearn using the trivial/toy `df` data and get the training set error:

```
In [9]: from sklearn.naive_bayes import GaussianNB
        import numpy as np

        X = df.values
        y = [0, 1, 1] # assume document classes
        sknb = GaussianNB()
        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 `defaultdict(int)` so that unknown words get mapped to value/index 0. Finally, function `vectorize()` will convert that to a 2D matrix, one row per document:

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

1.3.3 Estimating probabilities

To train our model, we need to estimate $P(c)$ and $P(c|d)$ 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 is not a unique word 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:

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

where $n_w(d)$ is the number of times w occurs 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:

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

1.3.4 Making predictions

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

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

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 it is monotonic and won't affect the *argmax*:

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

Or,

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

Avoiding log(0) If word w does not exist in class c (but is in V), then the classifier will try to evaluate $log(0)$, which gets an error. To solve the problem, we use *Laplace Smoothing*, which just means adding 1 to each word count in $n_w(d)$ when computing $P(w|c)$ and making sure to compensate by adding $|V|$ to the denominator (adding 1 for each vocabulary word:

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

where $|V|$ is the size of the vocabulary for all documents in all classes. Adding this to the denominator, keeps the $P(w|c)$ ratio the same. This way, even if $count(w,c)$ is 0, this ratio > 0.

(Note: Each doc's word vector is size $|V|$. During training, any w not found in docs of c, will have word count 0. Summing these gets us just total number of words in c. However, when we add +1, then c looks like it has every word in V. Hence, we must divide by $|V|$ not $|V_c|$.)

In your project, we can deal with both the Laplace smoothing by adding one to the data frame or 2-D matrix:

```
In [27]: df_ = df+1; df_
```

```
Out[27]:    cat  food  hong  kong
         0    4     2     1     1
         1    1     1     3     3
         2    2     1     2     2
```

and then computing the class word counts from this incremented matrix. The last two rows are from category 1, the positive reviews so we can isolate those and compute word counts per class:

```
In [29]: vpos = df_.iloc[1:3,:]; vpos
```

```
Out[29]:    cat  food  hong  kong
         1    1     1     3     3
         2    2     1     2     2
```

The word counts for documents in the positive category are then found using matrix operation `sum`:

```
In [36]: pos_word_counts = vpos.sum(axis=0); pos_word_counts
```

```
Out[36]: cat     3
         food    2
         hong    5
         kong    5
         dtype: int64
```

The $P(w|c)$ for all w is then:

```
In [43]: pos_total_words = np.sum(pos_word_counts)
         print(f"Total words in pos docs: {pos_total_words}")
         print(f"P(w|c):")
         pos_word_counts / pos_total_words
```

```
Total words in pos docs: 15
P(w|c):


Out[43]: cat     0.200000
         food    0.133333
         hong    0.333333
         kong    0.333333
         dtype: float64
```

It's important that you get used to using these vector operations for productivity and performance reasons.

1.3.5 Dealing with missing words

There's one last problem. If a future unknown document contains a word not in V (i.e., not in the training data), then $count(w, c)$ will be zero, which causes the same problem we just looked at. To fix that, all we have to do is add 1 to the denominator:

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

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:

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

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 $|V| + 1$ and each class will also have $|V| + 1$ words since no word has 0 word count. Each word vector will be of length $|V| + 1$.

When we count the words per class with `vneg.sum(axis=0)` and `vpos.sum(axis=0)`, these sums will include the "+1" we added for Laplace smoothing. Since we have augmented the vocabulary for unknown words at index 0, this will also increase `word_count_per_class` values since we added "+1" to word index 0 as well with `df+1`. So the formula for estimating $P(w|c)$ remains:

```
pos_word_counts / pos_total_words
```

To make a prediction for an unknown document, d, you will be given a feature vector composed of the word counts from d. Sum the multiplication of the word counts for $w \in d$ by the log of $P(w|c)$ for class c to get the weighted sum, then add the log of the class likelihood $P(c)$.

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 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`. My implementation is twice as fast as sklearn and appears to be more accurate for 4-fold cross validation.

1.4 Deliverables

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

- `load_docs(docs_dirname)`
- `vocab(neg, pos)`
- `vectorize_docs(docs, V)`
- `kfold_CV(model, X, y, k=4)`

and implement class `NaiveBayes621` with these methods

- `__init__(self)` (The body of this function is just keyword `pass`)
- `fit(self, X, y)`
- `predict(self, X)`

Please do not add the data to your repository!

1.5 Submission

In your github `bayes-`*userid* repository, you should submit your `bayes.py` file in the root directory. It should not have a main program that runs when the file is imported.

Please do not add data files to your repository!

1.6 Evaluation

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

```
$ python -m pytest -v test_bayes.py
============================================= test session starts ============================
platform darwin -- Python 3.7.1, pytest-4.0.2, py-1.7.0, pluggy-0.8.0 -- ...
cachedir: .pytest_cache
rootdir: /Users/parrt/courses/msds621-private/projects/bayes, inifile:
plugins: remotedata-0.3.1, openfiles-0.3.1, doctestplus-0.2.0, arraydiff-0.3
collected 6 items

test_bayes.py::test_load PASSED
test_bayes.py::test_vocab PASSED
test_bayes.py::test_vectorize PASSED
test_bayes.py::test_training_error PASSED
test_bayes.py::test_kfold_621 PASSED
test_bayes.py::test_kfold_sklearn_vs_621 PASSED

========================================== 6 passed in 21.04 seconds ========================
```

Notice that it takes about 20 seconds. If your project takes more than one minute, I will take off 10 points from 100. Each test is evaluated in a binary fashion: it either works or it does not. Each failed test cost you 15 points.