CS 8673 Programming Assignment 2: Naïve Bayes and k-means

September 22, 2017

This programming assignment is due on Wednesday, October 11, 2017, at 11:59pm.

Honor code note: You are welcome to discuss the assignment with other students, but you must individually and independently write the code and other deliverables that you submit for the assignment. Any copying of code from outside sources or other students will be considered an honor code violation.

---

1 Introduction

You have been provided:

- `PA3_Handout.pdf` - This file that you are reading
- `pa3data` - This is a folder containing the data files you will use for this assignment.

This assignment will continue your exposure to some simple machine learning algorithms. This assignment uses the same programming environment (numpy, scipy, matplotlib) as the previous one. Again, if you prefer to work in matlab or octave you are welcome to do so.

1.1 Credit

These assignments are based off of assignments from Andrew Ng's Stanford class, which you can find at:

http://openclassroom.stanford.edu/MainFolder/CoursePage.php?course=MachineLearning

Instructions there are for matlab/octave, so please reference them if you want to work in matlab. Solutions are available online for matlab, but please make sure that any code you submit is your own. The data for this assignment has been modified from the data used in that course, so the answers are different.

1.2 Deliverables

Please turn in your submission as a *single `.pdf` file* with answers to all of the relevant questions as well as the plots that you need and a code appendix with your solutions. This will make the grading process much easier.

1.3 Data

Download `pa3data.zip`, and extract the files from the zip file. They should end up in a folder called `pa3data`. This data bundle contains one set of data, for Naïve Bayes (part A), and two images, for k-means (part B).

1.4 Code Imports

All of the code examples assume that the following import statements have been executed (put at the top of your python file).

```
import numpy as np
from scipy import misc
from scipy import sparse as sps
import matplotlib.pyplot as plt
```

2 Part A: Naïve Bayes

In this exercise, you will use Naïve Bayes to classify email messages into spam and nonspam groups. Your dataset is a preprocessed subset of the Ling-Spam Dataset, provided by Ion Androutsopoulos. It is based on 960 real email messages from a linguistics mailing list.

2.1 Data Description

The dataset you will be working with is split into two subsets: a 700-email subset for training and a 260-email subset for testing. Each of the training and testing subsets contain 50% spam messages and 50% nonspam messages. Additionally, the emails have been preprocessed in the following ways:

1. **Stop word removal**: Certain words like “and”, “the”, and “of,” are very common in all English sentences and are not very meaningful in deciding spam/nonspam status, so these words have been removed from the emails.

2. **Lemmatization**: Words that have the same meaning but different endings have been adjusted so that they all have the same form. For example, “include”, “includes,” and “included,” would all be represented as “include.” All words in the email body have also been converted to lower case.

3. **Removal of non-words**: Numbers and punctuation have both been removed. All white spaces (tabs, newlines, spaces) have all been trimmed to a single space character.

As an example, here are some messages before and after preprocessing:

Nonspam message "5-1361msg1" before preprocessing

> Subject: Re: 5.1344 Native speaker intuitions
>
> The discussion on native speaker intuitions has been extremely interesting, but I worry that my brief intervention may have muddied the waters. I take it that there are a number of separable issues. The first is the extent to which a native speaker is likely to judge a lexical string as grammatical or ungrammatical per se. The second is concerned with the relationships between syntax and interpretation (although even here the distinction may not be entirely clear cut).

Nonspam message "5-1361msg1" after preprocessing

> re native speaker intuition discussion native speaker intuition extremely interest worry brief intervention muddy waters number separable issue first extent native speaker likely judge lexical string grammatical ungrammatical per se second concern relationship between syntax interpretation although even here distinction entirely clear cut

For comparison, here is a preprocessed spam message:

Spam message "spmsgc19" after preprocessing

> financial freedom follow financial freedom work ethic extraordinary desire earn least per month work home special skills experience required train personal support need ensure success legitimate home-based income opportunity put back control finance life ve try opportunity past fail live promise

Preprocessing has left occasional word fragments and nonwords. In the end, though, these details do not matter so much in our implementation.

2.2 Multinomial Naïve Bayes

To classify our email messages, we will use a multinomial Naïve Bayes model. The parameters of our model are as follows:

$$\phi_{k|y=1} = p(x_j = k|y = 1) = \frac{\left(\sum_{i=1}^{m}\sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = k \text{ and } y^{(i)} = 1\}\right) + 1}{\left(\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 1\} n_i\right) + |V|}$$

$$\phi_{k|y=0} = p(x_j = k|y = 0) = \frac{\left(\sum_{i=1}^{m} \sum_{i=1}^{m} \mathbb{1}\{x_j^{(i)} = k \text{ and } y^{(i)} = 0\}\right) + 1}{\left(\sum_{i=1}^{m} \mathbb{1}\{y^{(i)} = 0\} n_i\right) + |V|}$$

$$\phi_y = p(y = 1) = \frac{\sum_{i=1}^{m} 1\{y^{(i)} = 1\}}{m}$$

Recall from lecture that:

- $\phi_{k|y=1}$ estimates the probability that a particular word in a spam email will be the k-th word in the dictionary
- $\phi_{k|y=0}$ estimates the probability that a particular word in a nonspam email will be the k-th word in the dictionary
- ϕ_y estimates the probability that any particular email will be a spam email

Here are some other notation conventions:

- m is the number of emails in our training set
- The i-th email contains n_i words
- The entire dictionary contains $|V|$ words

You will calculate the parameters $\phi_{k|y=1}$, $\phi_{k|y=0}$, and ϕ_y from the training data. Then, to make a prediction on an unlabeled email, you will use the parameters to compare $p(x|y = 1)p(y = 1)$ and $p(x|y = 0)p(y = 0)$, as described in the class lectures. In this assignment, instead of comparing the probabilities directly, it is better to work with their logs. That is, you will classify an email as spam if you find

$$\log p(x|y = 1) + \log p(y = 1) > \log p(x|y = 0) + \log p(y = 0)$$

2.3 Implementing Naïve Bayes

In the data pack for this exercise, you will find a text file named "train-features.txt" that contains the features of emails to be used in training. The lines of this document have the following form:

```
2 977 2
2 1481 1
2 1549 1
```

The first number in a line denotes a document number, the second number indicates the ID of a dictionary word, and the third number is the number of occurrences of the word in the document. So in the snippet above, the first line says that Document 2 has two occurrences of word 977.

2.3.1 Load the features

Now load the training set labels and features into numpy in the following way:

```
# Load the labels for the training set
train_labels = np.loadtxt('train-labels.txt',dtype=int)

# Get the number of training examples from the number of
# labels
numTrainDocs = train_labels.shape[0]

# This is how many words we have in our dictionary
numTokens = 2500

# Load the training set feature information
M = np.loadtxt('train-features.txt',dtype=int)
# Create matrix of training data
train_matrix = sps.csr_matrix((M[:,2], (M[:,0], M[:,1])),  ...
           ... shape=(numTrainDocs,numTokens))
```

This first puts the y-labels for each of the m the documents into an $m \times 1$ vector. The ordering of the labels is the same as the ordering of the documents in the features matrix, i.e. the i-th label corresponds to the i-th row in `train_matrix`.

Second, this loads the data in our "train-features.txt" into a sparse matrix `train_matrix`, where each row of `train_matrix` represents one document in our training set, and each column represents a dictionary word. The individual elements represent the number of occurrences of a particular word in a document.

For example, if the element in the i-th row and the j-th column of `train_matrix` contains a "4", then the j-th word in the dictionary appears 4 times in the i-th document of our training set. Most entries in `train_matrix` will be zero, because one email includes only a small subset of the dictionary words.

2.3.2 A note on the features

In a Multinomial Naïve Bayes model, the formal definition of a feature vector $\vec{x}$ for a document says that $x_j = k$ if the j-th word in this document is the k-th word in the dictionary. This does not exactly match our numpy matrix layout, where the j-th term in a row (corresponding to a document) is the number of occurrences of the j-th dictionary word in that document.

Representing the features in the way we have allows us to have uniform rows whose lengths equal the size of the dictionary. On the other hand, in the formal Multinomial Naive Bayes definition, the feature $\vec{x}$ has a length that depends on the number of words in the email. We've taken the uniform-row approach because it makes the features easier to work with in numpy.

Though our representation does not contain contain any information about the position within an email that a certain word occupies, we do not lose anything relevant for our model. This is because our model assumes that each $\phi_{k|y}$ is the same for all positions of the email, so it's possible to calculate all the probabilities we need without knowing about these positions.

2.3.3 Training

You now have all the training data loaded into your program and are ready to begin training your data. Here are the recommended steps for proceeding:

1. Calculate ϕ_y
2. Calculate each $\phi_{k|y=1}$ for each dictionary word and store the all results in a vector.
3. Calculate each $\phi_{k|y=0}$ for each dictionary word and store the all results in a vector.

2.3.4 Testing

Now that you have calculated all the parameters of the model, you can use your model to make predictions on test data.

Load the test data in "test-labels.txt" and "test-features.txt" in the same way you loaded the training data. You should now have a test matrix of the same format as the training matrix you worked with earlier. The columns of the matrix still correspond to the same dictionary words. The only difference is that now the number of documents are different.

Using the model parameters you obtained from training, classify each test document as spam or non-spam. Here are some general steps you can take:

1. For each document in your test set, calculate $\log p(\vec{x}|y = 1) + \log p(y = 1)$
2. Similarly, calculate $\log p(\vec{x}|y = 0) + \log p(y = 0)$
3. Compare the two quantities from (1) and (2) above and make a decision about whether this email is spam. In numpy, you should store your predictions in a vector whose i-th entry indicates the spam/nonspam status of the i-th test document.

Once you have made your predictions, answer the questions in the deliverables section.

2.3.5 Note

Be sure you work with log probabilities in the way described in the earlier instructions and in the lecture videos. The numbers in this exercise are small enough that numpy will be susceptible to numerical underflow if you attempt to multiply the probabilities. By taking the log, you will be doing additions instead of multiplications, avoiding the underflow problem.

2.4 Part A: Deliverables

Please complete the following tasks and include requested answers in your project write up:

1. **Classification error** Compare your Naive-Bayes predictions on the test set to the correct labeling. How many documents did you misclassify? What percentage of the test set was this?

2. **Smaller training sets**

   Let's see how the classification error changes when you train on smaller training sets, but test on the same test set as before. So far you have been working with a 960-document training set. You will now modify your program to train on 50, 100, and 400 documents (the spam to nonspam ratio will still be one-to-one).

   You will see text documents in the data pack named "train-features-#.txt" and "train-labels-#.txt," where the "#" tells you how many documents make up these training sets. For each of the training set sizes, load the corresponding training data into your program and train your model. Then report the test error after testing on the same test set as before.

3. Be sure to include in your `.pdf` submission your code that you wrote for this assignment.

3 Part B: k-means

In this portion of the assignment, you will use k-means to compress an image by reducing the number of colors it contains.

3.1 Image Representation

The data pack for this exercise contains a 844-pixel by 537-pixel TIFF image named "b.tiff". It looks like the picture below.

In a straightforward 24-bit color representation of this image, each pixel is represented as three 8-bit numbers (ranging from 0 to 255) that specify red, green and blue intensity values. Our bird photo contains thousands of colors, but we'd like to reduce that number to 16. By making this reduction, it would be possible to represent the photo in a more efficient way by storing only the RGB values of the 16 colors present in the image.

In this exercise, you will use k-means to reduce the color count to various different k values. For example, when $k = 16$, you will compute 16 colors as the cluster centroids and replace each pixel in the image with its nearest cluster centroid color.

Because computing cluster centroids on a 844x537 image can be time-consuming, you will instead run k-means on the 254x162 image "b_small.tiff".

Once you have computed the cluster centroids on the small image, you will then use the 16 colors to replace the pixels in the large image.

3.2 k-means in numpy

In numpy, load the small image into your program with the following command:

```
A = misc.imread('b_small.tiff', mode='RGB')
```

This creates a three-dimensional matrix A whose first two indices identify a pixel position and whose last index represents red, green, or blue. For example, `A[50,33,3]` gives you the blue intensity of the pixel at position `y = 50, x = 33`. (The y-position is given first, but this does not matter so much in our example because the x and y dimensions have the same size).

Your task is to compute 16 cluster centroids from this image, with each centroid being a vector of length three that holds a set of RGB values. Here is the k-means algorithm as it applies to this problem:

3.2.1 The k-means algorithm

1. For initialization, sample 16 colors randomly from the original small picture. These are your k means $\mu_1, \mu_2, \ldots, \mu_k$.

2. Go through each pixel in the small image and calculate its nearest mean.

$$c^{(i)} := \arg\min_j \left\| x^{(i)} - \mu_j \right\|^2$$

3. Update the values of the means based on the pixels assigned to them.

$$\mu_j := \frac{\sum_i^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_i^m 1\{c^{(i)} = j\}}$$

4. Repeat steps 2 and 3 until convergence. This should take between 30 and 100 iterations. You can either run the loop for a preset maximum number of iterations, or you can decide to terminate the loop when the locations of the means are no longer changing by a significant amount or the cluster assignments of the pixels no longer changes.

Note: In Step 3, you should update a mean only if there are pixels assigned to it. Otherwise, you will see a divide-by-zero error. For example, it's possible that during initialization, two of the means will be initialized to the same color (i.e. black). Depending on your implementation, all of the pixels in the photo that are closest to that color may get assigned to one of the means, leaving the other mean with no assigned pixels. If this happens you can assign that cluster to a random pixel value as it's centroid.

3.2.2 Reassigning colors to the large image

After k-means has converged, load the large image into your program and replace each of its pixels with the nearest of the centroid colors you found from the small image.

When you have recalculated the large image, you can display and save it in the following way:

```
# Save out the modified image
plt.imshow(B)
plt.savefig('kmeans.png')
```

3.3 Part B Deliverables

Please complete the following tasks and include requested answers in your project write up:

1. Repeat the entire k-means process for $k = 2, \ldots 15$. Include the modified images after each run. How do the images differ from one another? Does the quality of the reproduction noticeably improve as k increases?

2. Select a single k between 2 and 15 and run the k-means process several times. Include the modified images from these runs. Does the algorithm find the same cluster centroids each time? Are any differences in centroid locations noticeable in the modified images?

3. If you were to run the code with $k = 1$ how many iterations would it take to converge? What would the single centroid correspond to?

4. Be sure to include in your `.pdf` submission your code that you wrote for this assignment.