B551 (online) Assignment 4: Machine Learning

Fall 2016
Due: Tuesday December 13, 11:59PM
(You may submit up to 48 hours late for a 10% penalty.)

In this assignment, we'll consider the problem of classifying text documents and images using various machine learning approaches.

You'll work in a group of 1-2 people for this assignment; we've already assigned you to a group (see details below) based on the feedback you provided. We tried to accommodate as many of your requests as possible, but we could not satisfy all of them. You should only submit **one** copy of the assignment for your team, through GitHub, as described below. All people on the team will receive the same grade on the assignment, except in unusual circumstances; we will collect feedback about how well your team functioned in order to detect these circumstances. Please read the instructions below carefully; we cannot accept any submissions that do not follow the instructions given here. Most importantly: please **start early,** and ask questions on Piazza or in office hours.

The following problems require you to write programs in Python. You may import standard Python modules for routines not related to AI, such as basic sorting algorithms and basic data structures like queues. You must write all of the rest of the code yourself. If you have any questions about this policy, please ask us. We recommend using the CS Linux machines (e.g. `burrow.soic.indiana.edu`). You may use another development platform (e.g. Windows), but make sure that your code works on the CS Linux machines before submission.

For each programming problem, please include a detailed comments section at the top of your code that describes: (1) a description of how you formulated the problem, including precisely defining the abstractions; (2) a brief description of how your program works; (3) a discussion of any problems, assumptions, simplifications, and/or design decisions you made; and (4) answers to any questions asked below in the assignment.

Academic integrity. You and your teammates may discuss the assignment with other people at a high level, e.g. discussing general strategies to solve the problem, talking about Python syntax and features, etc. You may also consult printed and/or online references, including books, tutorials, etc., but you must cite these materials (e.g. in source code comments). However, the work and code that you and your partners submit must be your group's own work, which your group personally designed and wrote. You may not share written answers or code with any other students except your own partner, nor may you possess code written by another student who is not your partner, either in whole or in part, regardless of format.

Part 0: Getting started

You can find your assigned teammate by logging into IU Github, at `http://github.iu.edu/`. In the upper left hand corner of the screen, you should see a pull-down menu. Select `cs-b551-fa2016`. Then in the yellow box to the right, you should see a repository called *userid1-userid2*-a4, where the other user ID corresponds to your teammate.

To get started, clone the github repository:

`git clone https://github.iu.edu/cs-b551-fa2016/`*your-repo-name*`-a4`

where *your-repo-name* is the one you found on the GitHub website above.

Part 1: Spam classification

Let's start by considering a straightforward document classification problem: predicting whether or not an e-mail is spam. We'll use a bag-of-words model, which means that we'll represent a document in terms of just an unordered "bag" of words instead of modeling anything about the grammatical structure of the document. In other words, a document can be modeled as simply a histogram over the words of the English language. If, for example, there are 100,000 words in the English language, then a document can be represented as a 100,000-dimensional vector, where the entries in the vector may correspond to binary values (1 if the word appears in the document and 0 otherwise), an integer (e.g., number of times the word appears in the document), or a real number (e.g., ratio of number of times the word appears in the document over total number of words in the document). Of course, most vectors will be sparse (most entries are zero).

1. First, implement a Naive Bayes classifier for this problem. For a given document D, we'll need to evaluate $P(S = 1|w_1, w_2, ..., w_n)$, the posterior probability that a document is spam given the features (words) in that document. Make the Naive Bayes assumption, which says that for any $i \neq j$, w_i is independent from w_j given S.

   Hint: It may be more convenient to evaluate the likelihood (or "odds") ratio of $\frac{P(S=1|w_1,...w_n)}{P(S=0|w_1,...,w_n)}$, and compare that to a threshold to decide if a document is spam or non-spam.

2. Implement a Decision Tree for this problem, building the tree using the entropy-based greedy algorithm we discussed in class.

To help you get started, we've provided a dataset in your repo of known spam and known non-spam emails, split into a training set and a testing set. For both the Bayesian and Decision Tree problems, train your model on the training data and measure performance on the testing data in terms of accuracy (percentage of documents correctly classified) and a confusion matrix. In general, a confusion matrix is an $n \times n$ table, where n is the number of classes ($n = 2$ in this case), and the entry at cell row i and column j should show the number of test exemplars whose correct label is i, but that were classified as j.

For each of the two models, consider two types of bag of words features: binary features which simply record if a word occurs in a document or not, and continuous features which record the frequency of the word in the document (as raw count, or a percentage, or some other function that you might invent). For the Bayesian classifier, your training program should output the top 10 words most associated with spam (i.e. the words with highest $P(S = 1|w)$) and the 10 words most associated with non-spam (words with highest $P(S = 0|w)$). For the decision tree, your training program should display a representation (e.g. a simple text visualization) of the top 4 layers of the tree.

Your program should accept command line arguments like this:

```
python spam.py mode technique dataset-directory model-file
```

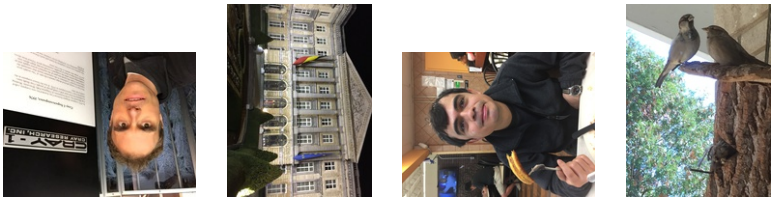
where *mode* is either `test` or `train`, *technique* is either `bayes` or `dt`, *dataset-directory* is a directory containing two subdirectories named `spam` and `notspam`, each filled with documents of the corresponding type, and *model-file* is the filename of your trained model. In training mode, *dataset-directory* will be the training dataset and the program should write the trained model to *model-file*, and in testing mode *dataset-directory* will be the testing dataset and your program should read its trained model from *model-file*. You can structure your model-file format however you'd like. We have not prepared skeleton code this time, so you may also prepare your source code however you'd like.

In your report, show results for two type of features with each of the two classifiers. Which techniques work best?

Part 2: Image classification

In this part we'll study a straightforward image classification task. These days, all modern digital cameras include a sensor that detects which way the camera is being held when a photo is taken. This metadata is then included in the image file, so that image organization programs know the correct orientation – i.e., which way is "up" in the image. But for photos scanned in from film or from older digital cameras, rotating images to be in the correct orientation must typically be done by hand.

Your task in this assignment is to create a classifier that decides the correct orientation of a given image, as shown in Figure 1.



180° 90° 270° 0°

Figure 1: Some sample images, and their correct orientation labels.

Data. To help you get started, we've prepared a dataset of images from the Flickr photo sharing website. The images were taken and uploaded by real users from across the world, making this a challenging task on a very realistic dataset.[1] Since image processing is beyond the scope of this class, we don't expect you to implement any special techniques related to images in particular. Instead, we'll simply treat the raw images as numerical feature vectors, on which we can then apply standard machine learning techniques. In particular, we'll take an $n \times m \times 3$ color image (the third dimension is because color images are stored as three separate planes – the red, green, and blue), and append all of the rows together to produce a single vector of size $1 \times 3mn$.

We've done this work for you already, so that you can treat the images simply as vectors and not have to worry about them being images at all. Your GitHub repo includes two ASCII text files, one for the training dataset and one for testing, that contain the feature vectors. To generate this file, we rescaled each image to a very tiny "micro-thumbnail" of 8×8 pixels, resulting in an $8 \times 8 \times 3 = 192$ dimensional feature vector. The text files have one row per image, where each row is formatted like:

```
photo_id correct_orientation r11 g11 b11 r12 g12 b12 ...
```

where:

- `photo_id` is a photo ID for the image.
- `correct_orientation` is 0, 90, 180, or 270. Note that some small percentage of these labels may be wrong because of noise in the data; this is just a fact of life when dealing with data from real-world sources.
- `r11` refers to the red pixel value at row 1 column 1, `r12` refers to red pixel at row 1 column 2, etc., each in the range 0-255.

[1] All images in this dataset are licensed under Creative Commons, and there are no copyright issues with using them for classroom purposes under fair use guidelines. However, copyrights are still held by the photographers in most cases; you should not post these images online or use these images for any commercial purposes without checking with the original photographer, who can be identified using the Flickr URL described on the next page.

Although you can get away with just using the above text files, you may want to inspect the original images themselves, for debugging or analysis purposes, or if you want to change something about the way we've created the feature vectors (e.g. experiment with larger or smaller "micro-thumbnails"). You can view the images in two different ways:

- You can view the original high-resolution image on Flickr.com by taking just the numeric portion of the photo_id in the file above (e.g. if the photo_id in the file is `test/123456.jpg`, just use 123456), and then visiting the following URL:

  `http://www.flickr.com/photo_zoom.gne?id=`*numeric_photo_id*

- We've provided a zip file of all the images in JPEG format, available via Canvas. We've reduced the size of each image to a 75×75 square, which still (usually) preserves enough information to figure out the image orientation. The ZIP file also includes UNIX scripts that will convert images in the zip file to the ASCII feature vector file format above. If you want, this lets you experiment with modifying the script to produce other feature vectors (e.g. smaller sized images, or in different color mappings, etc.) and to run your classifier on your own images.

The training dataset consists of about 10,000 images, while the test set contains about 1,000. For the training set, we've rotated each image 4 times to give four times as much training data, so there are about 40,000 lines in the train.txt file (the training images on Flickr and the ZIP file are all oriented correctly already). In test.txt, each image occurs just once (rotated to a random orientation) so there are only about 1,000 lines. If you view the test images on Flickr, they'll be oriented correctly, but the ones in the ZIP file are randomly oriented.

Step 1: Nearest neighbor. Implement a nearest neighbor classifier to predict the label of each test image. Write a program that is run from the command line like this:

```
python orient.py train_file.txt test_file.txt nearest
```

For each image to be classified, the program should find the "nearest" image in the training file, i.e. the one with the closest distance (least vector difference) in Euclidean space. It should then display the classification accuracy (in terms of percentage of correctly-classified images) as well as a confusion matrix, and should output a file called nearest_output.txt which indicates the estimated label for each image in the test file. The confusion matrix is a table with four rows and four columns. The entry at cell i, j in the table should show the number of test exemplars whose correct label is i, but that were classified as j (e.g. a "10" in the second column of the third row would mean that 10 test images were actually oriented at 180 degrees, but were incorrectly classified by your classifier as being at 90 degrees). A perfect classifier will have a diagonal confusion matrix, but we're unlikely to achieve that here. The nearest_output.txt file should coorespond to one test image per line, with the photo_id, a space, and then the estimated label, e.g.:

```
test/124567.jpg 180
test/8234732.jpg 0
```

Step 2: Adaboost. Implement a technique using decision stumps and Adaboost. Use very simple decision stumps that simply compare one entry in the image matrix to another, e.g. compare the red pixel at position 1,1 to the green pixel value at position 3,8. You can try all possible combinations (roughly 192^2) or randomly generate some pairs to try. Your program should be run like this:

```
python orient.py train_file.txt test_file.txt adaboost stump_count
```

and should train on the training file with the specified number of stumps, test on the testing file, and display the classification accuracy, confusion matrix, and output adaboost_output.txt, with the same format as in step 1.

Step 3: Neural network classification. Implement a fully-connected feed-forward network to classify image orientation, and implement the backpropagation algorithm to train the network using gradient descent. Your network should have one hidden layer (i.e. three layers total – the input layer, the hidden layer, and the output layer). Your program should be run like this:

```
python orient.py train_file.txt test_file.txt nnet hidden_count
```

and should train on the training file, using the specified number of hidden nodes, test on the testing file, and display the classification accuracy, confusion matrix, and output nnet_output.txt, with the same format as in step 1.

Step 4: Analysis and improvement. Each of the above machine learning techniques has a number of parameters and design decisions. For example, neural networks have network structure parameters (number of layers, number of nodes per hidden layer), as well as which activation function to use, whether to use traditional or stochastic gradient descent, which learning rate to use, etc. It would be impossible to try all possible combinations of all of these parameters, so identify a few parameters and conduct experiments to study their effect on the final classification accuracy. In your report, present neatly-organized tables or graphs showing classification accuracies and running times as a function of the parameters you choose. Which classifiers and which parameters would you recommend to a potential client? How does performance vary depending on the training dataset size, i.e. if you use just a fraction of the training data? Show a few sample images that were classified correctly and incorrectly. Do you see any patterns to the errors?

Finally, modify your code so that when run like this:

```
python orient.py train_file.txt test_file.txt best model_file
```

it uses whichever algorithm and parameter settings you recommend to give the best accuracy. The model_file is an optional parameter that you can choose to implement; if you'd like, your training routines can write their learned model to disk, so that when we run the "best" algorithm, training does not actually occur but instead a pre-trained model_file is loaded. Please use this option if your models take a long time to train (more than a few minutes). As in the past, a small percentage of your assignment grade will be based on how accurate your "best" algorithm is with respect to the rest of the class. We will use a separate test dataset, so make sure to avoid overfitting!

What to turn in

Turn in your source code file(s) as well as a **PDF document** containing your report for Part 1 and Part 2 (Step 4) by pushing to GitHub (remember to `add`, `commit`, `push`) — we'll grade whatever version you've put there as of 11:59PM on the due date. To make sure that the latest version of your work has been accepted by GitHub, you can log into the github.iu.edu website and browse the code online.

In addition to the information required in Part 1 and Part 2 (Step 4), either your report or the comments at the top of your source code should include a brief description of how your code works, and a discussion of any problems you faced, any assumptions, simplifications, and/or design decisions you made.