B551 Assignment 4: Learning

Fall 2016
Due: Thursday December 1, 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 using various machine learning approaches. You'll work in a group of 1-3 people for this assignment; we've already assigned you to a group (see details below) based on the input you provided. We tried to accommodate as many of your requests as possible, but we could not satisfy all of them (especially when you didn't fill out the form correctly or you and your candidate teammates gave conflicting answers!). 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-userid3*-a4, where the other user IDs correspond to your teammate(s).

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 least associated with spam (words with lowest $P(S = 0|w)$). 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:

```
./spam 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: Topic classification

Now let's consider a multi-class task: estimating the genre or topic of a document. To get you started, we've prepared a dataset of documents organized into twenty broad topics (e.g. religion, finance, etc.). But there's a twist. It's often the case that you do not have as much labeled training data as you would like for a particular classification task, but you might have a larger set of *unlabeled* data. In the extreme case, you may not have any labeled data at all – an unsupervised learning problem.

Let's say we have a dataset of documents $\mathcal{D} = \{d_1, d_2, ..., d_n\}$, a set of topics $\mathcal{T} = \{t_1, t_2, ..., t_m\}$, and a set of words in the English language $\mathcal{W} = \{w_1, w_2, ..., w_o\}$. A given document $D \in \mathcal{D}$ in our corpus has a topic $T \in \mathcal{T}$ and a set of words $W \subseteq \mathcal{W}$. In our training set, we'll always be able to observe D (a document ID number) and the set of words W in the document, but may not have T. In the test set, we'll always be able to observe D and W but never T.

Suppose that our corpus can be modeled as a Bayes Net with random variables D, T, and $\{W_1, ..., W_o\}$, where $W_i \in \{0, 1\}$ indicates whether word $w_i \in \mathcal{W}$ appears in the document or not. Variable D is connected to T via a directed edge, and then T is connected to each W_i via a directed edge.

Write a program that accepts command line arguments like this:

```
./topics mode dataset-directory model-file [fraction]
```

where *mode* is either `test` or `train`, *dataset-directory* is a directory containing directories for each of the topics (you can assume there are exactly 20), and *model-file* is the filename of your trained model. In training mode, an additional parameter *fraction* should be a number between 0.0 and 1.0 indicating the fraction of labeled training examples that your training algorithm is allowed to see.

In training mode, *dataset-directory* will be the training dataset and the program should write the trained model to *model-file*. When your training program is loading in the training data, it should "flip a coin" to decide whether it is allowed to see the class label associated with a given document, where the probability of "yes" is the fraction indicated on the command line. When *fraction* is 1.0, then this is a fully-supervised learning problem that is almost identical to problem 1 of Part 1. When *fraction* is 0.0, the program sees no labels, which is a fully unsupervised problem. In addition to the trained model, your program should output the 10 words with highest $P(T = t_i|W_j)$ for each of the 20 topics to a file called `distinctive_words.txt`. 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.

Hint: To deal with missing data, adopt an iterative approach. Estimate values in the conditional probabilities tables using the portion of the data that is labeled (or randomly, if no data is labeled). Then, estimate the distribution over T for each document D using variable elimination. Now you have "hallucinated" labels for the training set and can re-estimate the conditional probability tables, which can then be used to re-estimate the topic distributions, and so on. Continue iterating until convergence or until you run out of time. :)

In testing mode, report performance of the algorithm in terms of accuracy and a confusion matrix. Note that this testing program is nearly functionally equivalent to question 1 of Part 1, so if you make that implementation general enough, you should not have much work to do here. (Similarly, much of the learning code will be very similar to that problem also.) In your report, show the accuracies as a function of *fraction*; particularly interesting cases are when *fraction* is 1 (fully supervised), 0 (unsupervised), or low (e.g. 0.1).

What to turn in

Turn in your source code files 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.