COSI134-PA1

September 7, 2016

1 COSI 134 PA 1: Naive Bayes Classifier

Due Date: 09/21/2016 11:59 p.m.

1.1 Project Description

You are to write a Naive Bayes classifier and apply it to a text classification task (predicting the genders of the authors of blog posts). You must implement training and classification methods for your classifier, design and implement features specific to the given corpus, and report on your results. You are to submit a zip or tarball to Latte containing your code, a 1-2 page report (as a PDF file), and a model file for your best-performing classifier (see below).

1.2 Background

A supervised text classifier such as Naive Bayes uses a trained statistical model to infer or predict a label (also called a 'class' or 'category') for an unlabeled document (or 'instance'), given a vector of features. Useful resources:

1. Lecture 3 handout
2. Chapter 7 of Speech and Language Processing, Dan Jurafsky and James H. Martin
3. Chapter 6 of NLTK book
4. Naive Bayes documentation in sklearn

   Note: The tarball distributed with this assignment includes the raw corpus data and a few Python modules to help you get started. The included code is reasonably well documented (please read it!) and has been tested, but if you find anything unclear or incorrect, please let us know right away. If you want to just ignore it and start from scratch, that's fine, but not recommended.

1.3 Corpora

In this assignment, the documents you are responsible for classifying are blog posts; the labels will be 'M' for 'male' or 'F' for 'female', designating the gender of the post's author.

1.3.1 Blog posts dataset

There are 3,232 blog posts in the supplied corpus, which should be enough data to achieve reasonable results (see below for specifics) without taking unduly long to train.

1.3.2 Names Corpus

In addition to the blog gender corpus, a corpus of personal names is also provided, each of which is labeled as 'male' or 'female'. There are 5,001 female names and 2,943 male names, making this corpus 'larger' than the blog corpus; but each document consists of just one word, which makes dealing with them substantially faster and easier.

> **Note:** You may find it helpful to develop your classifier first with the names corpus and then move on to the blog post, but accurately classifying names is not a requirement for this assignment.

1.3.3 API for handling Corpus

The stater code contains a *corpus.py* module which serves as an API for handling all kinds of data instances involved in our corpus.

Class *Corpus*: This abstract class acts as a list-like container of data instances, for different type of corpus, e.g. Names Corpus, Blog Corpus, we have provided the inherented classes, *NamesCorpus* and *BlogsCorpus* to handle loading the data. You don't have to extend this part throughout the project.

Class *Document*: This class represents data instance in the corpus, e.g. in names corpus it represents $(name, label)$ pair; in blogs post corpus it represents $(blog\_post, label)$ pair.

It only provides basic *features* function which just returns the blog post. The following code shows how we load the dataset:

```
In [7]: from __future__ import unicode_literals
        from corpus import BlogsCorpus, Document
        from test_naive_bayes import Name

        # here each blog is a Document object
        blogs = BlogsCorpus(document_class=Document)

        # here each name is a Name object
        names = BlogsCorpus(document_class=Name)
```

The *NamesCorpus* and *BlogsCorpus* classes are the only ones that you should need to use directly. You will be responsible for writing subclasses of *Document* that include *features* methods tailored to the blog gender corpus, like the examples in the test_naive_bayes module (classes *EvenOdd*, *BagOfWords* and *Name*).

1.4 The Naive Bayes classifier

The classifier module contains an abstract base *Classifier* class that you should use as a superclass for your Naive Bayes classifier. An example subclass called *TrivialClassifier* may be found in the *test_classifier* module; it doesn't do much, but it illustrates the basic interface. You should not need to change anything in either of those two modules.

Your classifier should go in the *naive_bayes* module, which is just a skeleton in the distributed code. Your job is to flesh out that skeleton. The included *test_naive_bayes* module includes some basic tests, including ones for baseline classifier performance; they will obviously fail at first, but should all pass by the time you're done.

> **Note:** You may (but need not) add your own modules to the ones provided. You may also (but need not) use any third-party libraries you wish (e.g., NLTK, NumPy), but you cannot just import (or copy!) a third-party Naive Bayes classifier; its implementation must be entirely your own.

1.5 Baseline

The tests in the distributed *test_naive_bayes* module check the accuracy (i.e., the proportion of correctly classified documents) for: 1. a 'pseudo-corpus' of integers trivially classified as 'even' or 'odd'. 2. Names gender Corpus 3. Blog posts gender Corpus

Your model should be able to achieve the following performance given the specified data split and *features* functions: 1. 100% for the integers, 2. 70% for the names corpus with the supplied features 3. 55% for the blogs corpus with bag-of-words as features

These are the baselines for your Naive Bayes model.

1.6 Improve the model

Just implementing a classifier is not sufficient—you have to show that it works, and then make it work better!

Once your classifier is working (i.e., it can pass those baseline tests), your next task is to improve its performance on the blog gender corpus. A state-of-the-art classifier (not Naive Bayes) with linguistically sophisticated features can achieve close to 90% accuracy (see the paper Improving Gender Classification of Blog Authors by Mukherjee and Liu). However, you are not required to beat this :)

Here are some methods you could try to improve your model:

- **Feature Engineering.** Feature engineering is the part that needs more about linguistic intuition and involves a lot of trial-and-error. Clearly organize your trained models (you can save it using the classifier's *save* method) and record how each feature set affects your performance. Start with some simple things first: experiment with tokenization, n-grams, frequency counts, etc.
- **Different smoothing techniques.** Smoothing gives us a more realistic distribution by allocating estimated probability to the unseen data. Try using different smoothing technique and measure the performance.
- **Bernoulli vs Multinomial.** In class, we have discussed the difference between the two variation of the Naive Bayes model. These two methods imply different assumptions about the real data distribution. Try comparing and analyzing the result.

1.6.1 Compare different evaluation metrics

Sometimes it's not enough to only have one metric for the task. You should utilize more measurements to get clear understanding of how your model perform.

We also included an imbalanced data set (˜90% of the data belongs to one class). When improving your model, evaluating the performance with both **accuracy** and **F-measure**, see how different metrics change. Plot and analyze what you have found.

> **Note:** you need to implement your own F-measure function. And since F-measure is a per-class measurement in our case. You could use unweighted (averaging) of F-measure score of both classes or use weighted one (weighted by the proportion of each class).

1.7 Grading

Your grade will be based on the correctness of your classifier's training and classification algorithms (60%), its performance on the blog gender corpus with your best set of features (10%), code clarity and style (10%), and the quality of your report (20%).